Diplomarbeit

Susanna Thuranszky
02.11.1978
Informatik im WS 2004/2005

Erklärung
gemäß §31 Abs. 5 RaPO

Ich erkläre hiermit, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

München, den 06.02.2005

1 Einführung

Sneed [Sne91: 24] beschreibt Software als ein Abbild eines Modells der realen Welt (siehe Abb. 1.1).

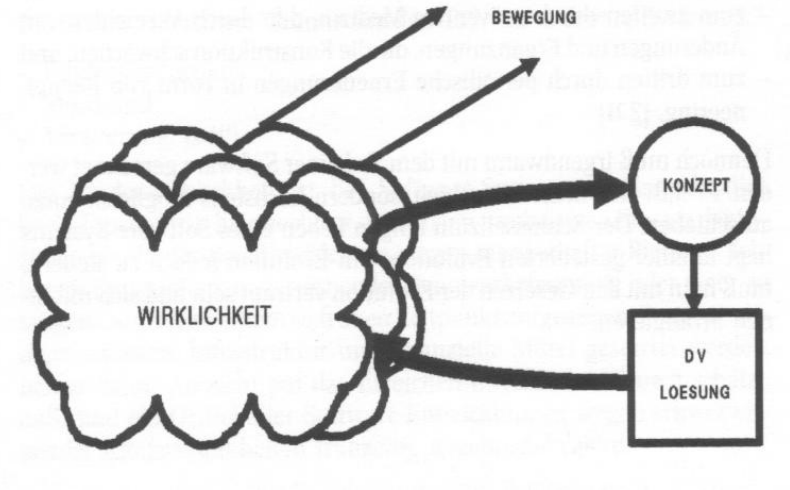


Abbildung 1.1 [Sne91: 66]

Datenkapseln sind Abbildungen von Datenobjekten, die ihrerseits Abbildungen echter Objekte der betrieblichen Wirklichkeit, so genannter „real word entities“, sind. Programme sind Abbildungen von Vorgängen, die ihrerseits Abbildungen echter Geschehnisse oder Geschäftsprozesse sind. Wenn Abläufe und Informationsbedürfnisse der echten Welt sich verändern, verändert sich auch das Modell dieser Welt und mit ihm die Software, die das Modell an einem digitalen Rechner realisiert. Software veraltet in dem Maße, wie sie mit der Wirklichkeit nicht Schritt hält [Sne91: 24]. Die Wartung hat somit die Aufgabe, die Modelle und somit auch die Software an die Wirklichkeit anzupassen.

Software-Entwickler stehen vor einem Berg von unstrukturierten, monolithischen, nicht normierten, unflexiblen, unportablen und unzuverlässigen Software-Systemen [Sne91: 19]. Martin und McClure vergleichen die Situation mit einem Eisberg (siehe Abbildung 1.2). Die Spitze des Eisbergs ist die Software-Entwicklung, die gut sichtbar ist, dagegen verbirgt sich die Software-Wartung unter der Oberfläche. Die Wartung der heute bestehenden Software braucht einen enormen Aufwand. Die Zeit und die Kosten dafür übersteigen nicht nur oft die Investitionen der Entwicklung, sondern sind auch noch unvorhersehbar, schwer planbar und bergen unangenehme Überraschungen.

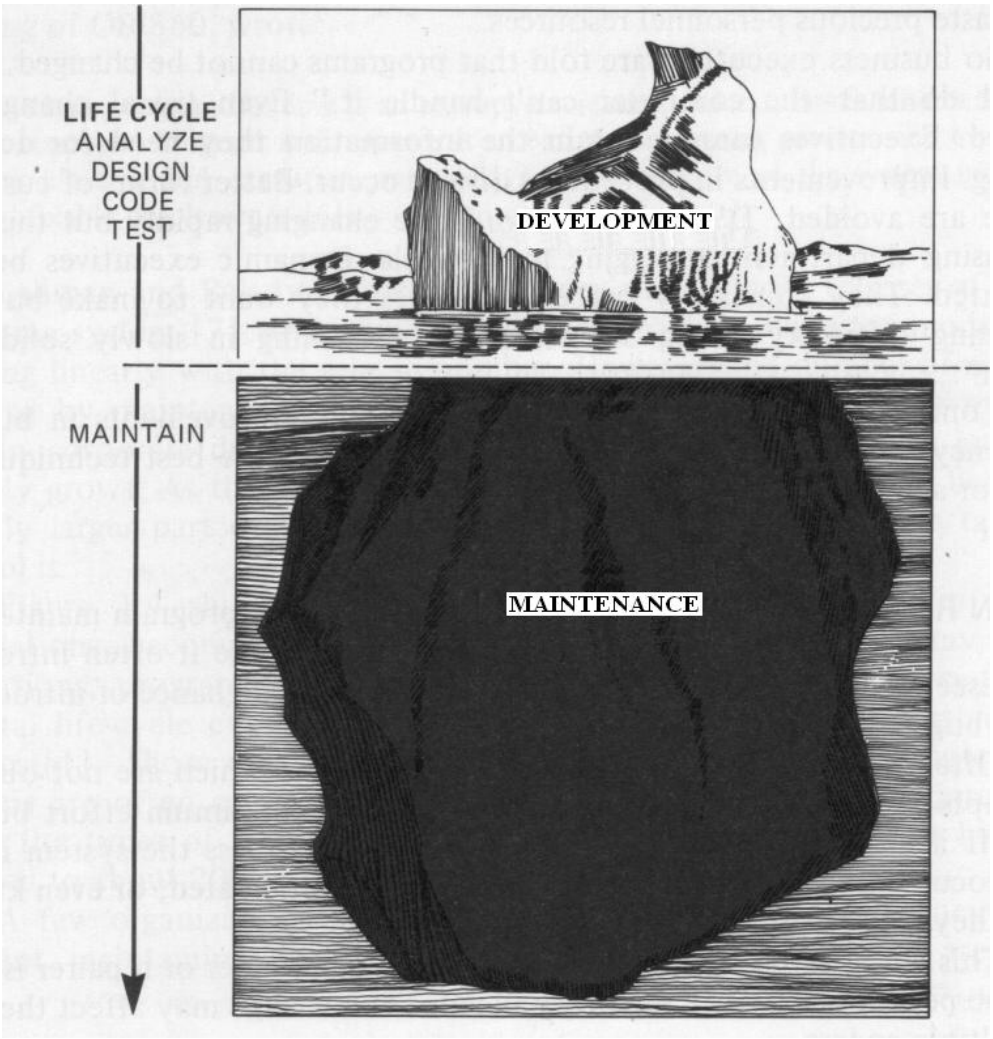


Abbildung 1.2 [Mar83: 7]

Es macht keinen Sinn, nach Schuldigen zu suchen. Die Software ist nun mal da, wird gebraucht und diese mit völlig neuer Software zu ersetzen, wäre in den meisten Fällen mit unübersehbaren Investitionen und Risiken verbunden, zu denen kaum jemand bereit ist. Es bleibt nur noch übrig, sich mit der Software-Wartung auseinander zu setzen und Lösungen für das bestehende Problem der bisher eingesetzten Systeme zu finden.

In dieser Diplomarbeit werden die Systeme aus konzeptioneller Sicht betrachtet. Ziel dieser Arbeit ist es, die Wartung speziell in Bezug auf konzeptionelle Aspekte zu thematisieren. Es wird gezeigt, dass Wartung bei Beschränkung auf die Codierungsebene indiskutabel ist und nur unter Einbeziehung restrukturierter Realitätsmodelle zu sinnvollen Ergebnissen führt. Die Verlagerung der Wartungstätigkeiten auf höhere Ebenen, sprich Korrekturen und Änderungen auf der Spezifikationsebene durchzuführen, ermöglicht es, die Problematik des Software-Eisbergs zu reduzieren.

Zuerst wird in Kapitel 2 der Begriff „Wartung“ definiert. Nach der Definition sollen in Kapitel 3 die Ursachen dafür vorgestellt werden. Hier werden die Anlässe aufgezählt, die eine Software-Wartung erforderlich machen. Leider hat die Wartung mit einem Berg von Problemen zu kämpfen. Kapitel 4 ist die Zusammenfassung der Probleme, mit der die Software-Wartung konfrontiert ist. Danach werden die Ziele der Wartung in Kapitel 5 vorgestellt. Diese können anhand der Definition, der Ursachen sowie der Probleme der Wartung, also anhand Kapitel 2, 3 und 4, dargelegt werden. Der wichtigste Beitrag dieser Arbeit ist in Kapitel 6 zu finden. Hier wird beschrieben, wie Software-Wartung bisher war, was falsch gemacht worden ist und wie in Zukunft vorgegangen werden sollte. Es wird über Wartungsansätze berichtet, die zu einer Lösung der beschriebenen Probleme beitragen können. Zum Schluss werden in Kapitel 7 konzeptionelle Ideen zur Software-Entwicklung aufgeführt und diese, anhand einiger Anwendungsbeispiele, besprochen.

2 Definition der Software-Wartung

In diesem Kapitel sollen die in der Literatur zu findenden Definitionen der Software-Wartung zitiert werden. Die Definitionen in 2.1 beschreiben die Software-Wartung aus verschiedenen Sichten. Eine weitere eigene Definition würde an dieser Stelle zu keinen neuen Erkenntnissen führen. Es werden jedoch die Definitionen miteinander verglichen und nach zwei Aspekten kategorisiert. Zuerst kommen in 2.1.1 Autoren durch Zitate zu Wort, die Software-Wartung so allgemein halten, dass eine Kategorisierung kaum möglich wäre. Anschließend werden die Definitionen vorgestellt, die Software-Wartung aus einer spezifischen Sicht betrachten. Manche Autoren konzentrieren sich auf die technischen Aspekte der Software-Wartung. Eine Definition wird in 2.1.2 aufgeführt. Danach folgt in 2.1.3 eine Definition, die die Wartung nach konzeptionellen Aspekten beschreibt. Eine umfassende Begriffserklärung bietet der ANSI-Standard. Diese beinhaltet sowohl die technische wie auch die konzeptionelle Sicht und wird in 2.1.4 zitiert. Die Definition kommt sehr nah an eine Aufteilung der Wartungstätigkeiten heran. Um diesen Gedanken weiterzuführen, stehen in 2.2 die Definitionen, die bewusst eine Einteilung vornehmen. Auch hier sollen die Tätigkeitsbereiche nach technischen und konzeptionellen Aspekten beurteilt werden.

2.1 Definitionen

Hier wird der Begriff Software-Wartung erklärt. In 2.1.1 stehen Definitionen, die eine sehr allgemeine Beschreibung bieten. Danach folgen in 2.1.2 und 2.1.3 Zitate, die Software-Wartung aus einer technischen bzw. konzeptionellen Sicht betrachten. In 2.1.4 ist eine Definition zu finden, die sowohl die technische wie auch die konzeptionelle Sicht einbezieht.

2.1.1 Allgemeine Sicht

Thurner schreibt, dass „Software-Wartung der gesamte Aufwand ist, der nach der Übergabe oder Einführung eines Systems auftritt" (zitiert nach [Leh91: 3]).

Auch Brun beschreibt die Software-Wartung sehr allgemein [Bru81: 135]:

> „Unter Wartung werden all jene Tätigkeiten verstanden, die nach Einführung eines Systems dessen Verfügbarkeit erhöhen oder Lebensdauer verlängern."

Bei Bischoff steht ebenfalls eine sehr allgemein gehaltene Definition [Bis87 zitiert nach [Leh91: 4]]:

> „Wartung ist die Modifikation eines Softwareproduktes nach Übergabe bzw. Lieferung."

2.1.2 Technische Sicht

Heinrich und Burgholzer [Hei90 zitiert nach [Leh91: 3]] treffen eine konkretere Aussage. Deren Definition konzentriert sich auf die technische Seite der Wartung:

> „Die Gesamtheit aller Maßnahmen zur Erhaltung oder Wiederherstellung der Funktionsfähigkeit und der Leistungsfähigkeit der Betriebsmittel der Informationsfunktion, insbesondere der Hardware und der Software".

2.1.3 Konzeptionelle Sicht

Riggs [Rig82 zitiert nach [Leh91: 4]] beschreibt dagegen die Wartung mehr nach konzeptionellen Aspekten:

> „Maintenance is the activity associated with keeping operational computer systems continously in tune with the requirements of users, data processing operations, associated clerical functions and external demands from governmental or other agencies."

2.1.4 Technische und konzeptionelle Sicht

Der ANSI-Standard bietet eine umfassende Definition, in der sowohl technische, wie auch konzeptionelle Aspekte enthalten sind:

> „Modifications of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment" (ANSI Standard Nr. 789, zitiert nach [Leh91: 4]).

2.2 Wartung nach Tätigkeitsbereichen

Wenn es um die Definition der Wartung geht, findet man in der Literatur oft eine Einteilung der Wartungstätigkeiten. Auch bei dieser können technische sowie konzeptionelle Aspekte deutlich werden. Curth und Giebel [Cur89: 42, 80] weisen auf die klassische Definition von Lientz und Swanson hin (Lientz, B.P.; Swanson, E. B.: Maintenance, 1980). Demnach ist die Rede von Fehlerbehebung (**corrective maintenance**), Anpassung der Funktionalitäten an veränderte Anforderungen (**adaptive maintenance**) und Weiterentwicklung des bestehenden Systems (**perfective maintenance**). An vielen anderen Stellen findet man diese Einteilung, wobei die Autoren sich einen gewissen Freiraum erlauben und die einzelnen Bereiche somit nicht genau gleich erklären. Dennoch bleibt festzustellen, dass bei dieser Einteilung die korrektive Wartung sich auf rein technische Aspekte bezieht. Dagegen kann perfektive und adaptive Wartung sowohl technische, als auch konzeptionelle Gesichtspunkte beinhalten.
Auch Swanson nimmt diese Einteilung der Wartungsaktivitäten vor [zitiert nach [Leh91: 4]]. Er erklärt die Begriffe auf folgende Weise:

> „corrective maintenance, bei der es sich um die eigentliche Notfallwartung handelt, d.h. um die Korrekturen zunächst unerkannter Fehler, die beim Einsatz des Anwendungssystems auftreten.
>
> perfective maintenance, d.h. eine Erweiterung des Funktionsumfangs und eine Verbesserung der Leistungen.
>
> adaptive maintenance, d.h. Anpassung des Anwendungssystems an eine veränderte Umwelt wie z.B. Betriebssystemänderungen, neue Hardware oder geänderte gesetzliche Bestimmungen."

Auch Kroha teilt Software-Wartung nach Aktivitäten auf [Kro97: 181]. Er unterscheidet aber nicht nur korrektive, perfektive und adaptive sondern auch präventive Wartung:

> „Korrektive Wartung (etwa 22% der gesamten Wartung)
> Der Zweck der korrektiven Wartung ist die Behebung von Fehlern, die im Software-Produkt aus den vorhergehenden Entwicklungsphasen geblieben sind.

Perfektive Wartung (etwa 42% der gesamten Wartung)

Das Ziel der perfektiven Wartung ist es, nach Erfahrungen mit dem Betrieb des Software-Systems dessen Eigenschaften zu verbessern. Es handelt sich z.B. um Leistung, Speicheransprüche, Verständlichkeit der Benutzerschnittstelle und andere Parameter des Systems, die erst nach einer längeren Betriebszeit sichtbar werden.

Adaptive Wartung (etwa 23% der gesamten Wartung)

Die adaptive Wartung bedeutet die Anpassung des Software-Produkts an Änderungen der Anforderungen. Dazu gehören die Gesetzes- und Vorschriftenänderungen, Hardware- und Software-Plattform-Änderungen und auch geänderte Wünsche und Ansprüche der Kunden.

Präventive Wartung (etwa 13% der gesamten Wartung)

Zu der präventiven Wartung gehören Änderungen, die zwar momentan nicht unbedingt notwendig sind, die jedoch die künftige Wartung verbessern, z.B. Umstrukturierung, Änderungen der Software-Architektur. Es ist eine Vorsorge und Vorbereitung für das Reengineering."

3 Ursachen der Software-Wartung

Schon in der Software-Entwicklungsphase können Ursachen für Wartung begründet sein. Noch bevor eine Software fertig gestellt wird, können Planungsfehler, ein falsches Konzept, nicht geeignete Architektur den Ausgangspunkt für eine spätere Wartung bilden. Sneed bezeichnet diese Probleme als Kinderkrankheiten der Software, wie z.B. „mangelhafte Planung, fehlende Kapazitäten, unzureichende Benutzerbeteiligung, unadäquate Infrastruktur, …" [Sne91: 64]. Aber auch später in der Einsatzphase entstehen weitere Gründe für eine Wartung. In der Literatur werden die Wartungsgründe nach verschiedenen Aspekten kategorisiert. Es wird in 3.1 ein Vorschlag für die Kategorisierung vorgestellt. Hierbei wird eine mögliche Ursache, die Fehlerbehebung, ausführlich behandelt. Danach werden in 3.2 die weiteren Wartungsursachen benannt. In 3.2.1 stehen Ergebnisse von wissenschaftlichen Arbeiten. Hier sind die Ursachen nach prozentualer Verteilung gruppiert, eine weitere Systematisierung unterbleibt. In 3.2.2 sollen die Ursachen nach zwei Aspekten aufgeteilt werden. Diejenigen, die auf technischer Basis beruhen, sind in 3.2.2.1 zusammengefasst. Des Weiteren sind konzeptionelle Aspekte in 3.2.2.2 angeführt, wobei diese in interne in 3.2.2.2.1 und externe Ursachen in 3.2.2.2.2 aufgeteilt sind. Zum Schluss werden in 3.2.3 die verschiedenen Ursachen bzw. deren Einteilung ausgewertet.

3.1 Kategorien der Wartungsursachen

Wartungsursachen können nach den verschiedensten Aspekten in Kategorien aufgeteilt werden. Eine mögliche Aufteilung nimmt Heinemann vor. Er schlägt den folgenden Systematisierungsansatz vor (Abbildung 3.1):

Wartungsursachen			
Menschliche Faktoren		Dynamik der Umwelt	
Fehler bei der Entwicklung	Fehler in der Wartung	Veränderungen innerhalb des DV-Systems	Veränderungen außerhalb des DV-Systems

Abbildung 3.1 [Hei87: 29]

In Abbildung 3.1 werden unter menschlichen Faktoren als Wartungsursachen Fehler aufgezählt, die einerseits in der Entwicklung, andererseits in der Wartung liegen können.

Abbildung 3.2 zeigt die Aufteilung der Fehler auf die einzelnen Entwicklungsphasen. Offensichtlich ist, dass der überwiegende Anteil der Fehler bereits in der Entwurfsphase begründet ist. Die Mehrzahl der Fehler wird allerdings erst in der Einsatzphase entdeckt. Die Wartung ist nun für die Behebung der Fehler verantwortlich, obwohl meist keine unmittelbare Einflussnahme auf die Fehlerentstehung möglich ist [Leh91: 6].

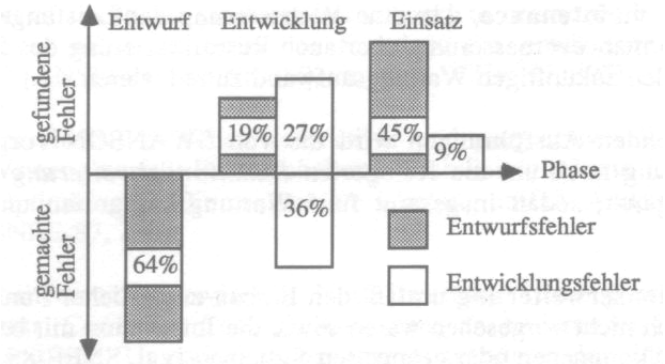


Abbildung 3.2 [Leh91: 6]

Lientz und Swanson haben die Verteilung der Wartungstätigkeiten untersucht und dabei herausgefunden, dass die Fehlerbehebung nur 17% der Wartungsaktivitäten ausmacht.

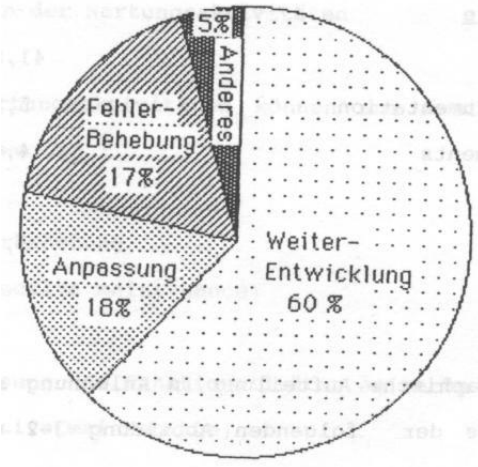


Abbildung 3.3 [Cur89: 44]

Aus der Abbildung 3.3 ist klar zu erkennen, dass sich der heutige Wartungsbegriff aus einer Vielzahl von Aktivitäten zusammensetzt und die bloße Fehlerbehebung nur einen geringen Anteil ausmacht. Die restlichen 83% der Wartung können aus verschiedenen Ursachen entstehen. Nach Heinemann sind diese auf die Dynamik der Umwelt zurückzuführen. In dem folgenden Kapitel werden die Ursachen aufgezählt.

3.2 Aufzählung der Wartungsursachen

Die Ursachen, die eine Watung erforderlich machen, können nach verschiedenen Kriterien aufgelistet werden. Eine mögliche Kategorisierung wurde in 3.1 vorgeschlagen. Im Folgenden werden die Wartungsursachen jedoch nach andern Prinzipien erfasst. In 3.2.1 werden Untersuchungen gezeigt, die Wartungsursachen nach prozentualer Verteilung aufzählen. Danach werden in 3.2.2 die Ursachen nach zwei Aspekten gruppiert. Ich entschied mich, ähnlich wie schon bei der Definition der Wartung, technische und konzeptionelle Aspekte zu trennen. Ursachen, die auf technische Hintergründe zurückzuführen sind, werden in 3.2.2.1 aufgeführt. Konzeptionelle Aspekte sind in 3.2.2.2 zu finden. Hier wird eine Unterscheidung zwischen internen Ursachen in 3.2.2.2.1 und externen in 3.2.2.2.2 vorgenommen. Zum Schluss erfolgt in 3.2.3 eine Auswertung der Aufzählung und Kategorisierung der Ursachen.

3.2.1 Ursachen nach Häufigkeit

Budde untersuchte die Gründe für Wartung anhand seiner Umfrage. Er kategorisierte diese nicht nach Wartungstätigkeiten. Die Ursachen für Änderungen bei den Anwendungssystemen werden nach prozentualer Verteilung aufgeschlüsselt [Bud80 zitiert nach [Leh91: 8]]:

Markt/Produkt	42%
Spätere Wünsche von Fachabteilungen	30%
Planungsfehler	9%
Pogrammfehler	6%
DV-interne Verbesserungen	5%
Gesetzgebung	5%
Umstellungen	3%

Auch Lientz und Swanson untersuchten die Wartungsursachen. In ihrer umfangreichen Studie gelangen sie zu einer anderen Aufteilung [Lien80 zitiert nach [Leh91: 8]]:

Verbesserungen für Benutzer	41,8%
Änderungen der Eingabedateien	17,4%
Notkorrekturen	12,4%
Routinekorrekturen	9,3%
Änderungen der Systemumgebung	6,2%
Dokumentationsverbesserung	5,5%
Codeoptimierung	4,0%
Sonstige Änderungen	3,4%

3.2.2 Ursachen nach Kategorien

Hier folgt eine Aufteilung der Wartungsursachen nach zwei Kriterien. Wartung kann aus einer technischen oder einer konzeptionellen Sicht betrachtet werden. In den zwei Gruppen werden die einzelnen Ursachen aufgezählt. Die Aufzählung kann aufgrund der Vielzahl situativer Bedingungen immer nur unvollständig sein. Die hier aufgeführten Beispiele sollten deshalb lediglich als ein Auszug potentieller Wartungsursachen verstanden werden.

3.2.2.1 Technische Aspekte

Hierzu gehören jegliche Ursachen, die aus einem technischen Aspekt heraus entstehen. Bei der Aufzählung spielt die Reihenfolge keine Rolle. Eine Erweiterung ist je nach Spezialbeispielen beliebig möglich. Aus einer technischen Betrachtungsweise können die folgenden häufigen Beispiele aufgezählt werden (vgl. [Kli99: 9]):

- Anpassung der Hardware
- Anpassung der Software, die mit dem Programm in Beziehung steht
- Korrektur von Softwarefehlern/Programmfehlern
- Code Reviews, Code-Inspektionen, Code-Optimierung
- Modernisierung, Instandsetzung
- Steigende Anforderung an die Produktionstechnik
- Ungeplante Wiederverwendung und Programmportierung

3.2.2.2 Konzeptionelle Aspekte

Ursachen aus konzeptionellem Blickwinkel können in zwei Gruppen eingeteilt werden. Aus der Sicht des Unternehmens können die Ursachen entweder intern oder extern sein. Beispiele für interne Ursachen sind Veränderungen der Systemumgebung, der Unternehmens-organisation, der Geschäftsprozesse, der Benutzeranforderungen. Extern können Veränderung der Rahmenbedingungen, der Konkurrenzdruck, Gesetzesänderungen, Änderungen des Marktes und Produktes Ursachen von Wartungsmaßnahmen sein. Hier könnten selbstverständlich auch noch andere Beispiele aufgelistet werden.

3.2.2.2.1 Interne Ursachen

An dieser Stelle werden die internen Ursachen aus der Sicht des Unternehmens, wie Veränderungen der Systemumgebung, der Unternehmensorganisation, der Geschäftsprozesse, der Benutzeranforderungen, ausführlich erklärt und mit Beiträgen aus der Literatur wie folgt unterstützt:

- Veränderung der Systemumgebung:
  Heisenberg fasst die Gründe für die Änderung der Umwelt mit dem Unsicherheitsprinzip zusammen, nachdem jedes System die Umgebung, aus der es hervorgegangen ist, verändert und damit eine neue Umgebung schafft [Sne92: 26]. Belady und Lehman begründen die Änderungen mit dem Gesetz der kontinuierlichen Veränderung (zitiert nach [Hei87: 73]): „A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it." Eisner weist auf das Systemalter hin und schreibt, dass die Verbindung des Software-Systems zur Außenwelt einem „Alterungs-Prozess" unterworfen ist und dass diese Verbindung bei der Wartung aufgefrischt werden muss [Eis88: 45].

- Veränderung der Unternehmensorganisation:
  Die Struktur und der Führungsstil kommerzieller und administrativer Organisationen weisen im Zeitablauf mehr oder weniger starke Veränderungen auf. Die Wartung hat die Aufgabe, das DV-System so anzupassen, dass es den veränderten Anforderungen gerecht wird [Hei87: 41].

- Veränderung der Geschäftsprozesse:
  Die Geschäftsprozesse ändern sich und werden im DV-System nicht mehr korrekt abgebildet [Sie03: 165]. Somit unterstützt das System den Betrieb des Unternehmens nicht mehr so weit, wie das gewünscht wäre.

- Veränderung der Benutzeranforderungen:
  Häufig ist sich der zukünftige Benutzer am Anfang eines Projekts nicht genau darüber im Klaren, was er will. Sobald er sich aber mit dem fertigen Produkt konfrontiert sieht, erkennt er plötzlich bisher ungeahnte Änderungsmöglichkeiten [Fäs96: 11]. Auch spätere Änderungsanforderungen können aus verschiedenen Gründen entstehen, wie

z.B. Wünsche nach Effizienzverbesserungen, Funktionserweiterungen, Verbesserungen der Benutzeroberfläche [Hei87: 38].

3.2.2.2.2 Externe Ursachen

Im Folgenden sind die Ursachen aufgeführt, die aus der Sicht des Unternehmens externe Hintergründe haben. Die Beispiele sind mit Hinweisen aus der Literatur angereichert.

- Veränderung der Rahmenbedingungen:
  Programme unterstützen einen Ausschnitt der Realität. Wenn sich die Umwelt ändert, müssen die Programme der neuen Situation angepasst werden [Eis88: 45].

- Konkurrenzdruck:
  Die kommerziell-administrativen Software-Systeme sind wichtige Elemente des betrieblichen Informations-Systems. Als solche sind sie dem Konkurrenzdruck ausgesetzt – bessere und schnellere Informationen sind nötig, um Konkurrenzvorteile zu wahren oder neue Vorteile der Gegenseite wieder aufzuholen [Eis88: 45].

- Gesetzesänderungen:
  Merkmale einer dynamischen Umwelt können häufige Gesetzesänderungen mit Auswirkungen auf das Geschäftsvolumen (z.B. steuerliche Anreize für Wertpapiere im Bankbereich) sein [Leh91: 35]. Auch andere gesetzliche Bestimmungen können in dem Software-System implementiert sein und somit sind gesetzliche Veränderungen eine der Wartungsursachen [Hei87: 40].

- Änderungen des Marktes und Produktes:
  Veränderungen einer Unternehmung im Markt (Wechsel, Hinzufügen oder Entfernen von Produktlinien oder Änderungen im Vertriebssystem) können Systemanpassungen der Software in erheblichem Umfang nach sich ziehen [Hei87: 40].

3.2.3 Auswertung

Hier soll noch mal darauf hingewiesen werden, dass die Ursachen der Wartung außerordentlich vielschichtig sind. Sie können nur im genauen Kontext vollständig erfasst

und ausführlich veranschaulicht werden. Die oben vorgestellte Kategorisierung der Ursachen der Wartung kann daher nur als ein Vorschlag gelten und soll die Vielfalt der möglichen Ursachen andeuten. Als Ergebnis kann dennoch festgehalten werden, dass durch Verbesserungen in der Software-Entwicklung Wartungsursachen zum Teil erheblich erleichtert werden können, aber schlussendlich kann auf Wartung nicht verzichtet werden, denn Veränderungen der Realität sind nicht aufzuhalten. Eine Notwendigkeit der Wartung, die für immer erhalten bleibt, ergibt sich aus dem Grund, dass ein für immer vollkommenes IT-System nicht zu erschaffen ist. Sehr aufwendige Wartungsmaßnahmen tauchen besonders dann auf, wenn das grundsätzliche Programmkonzept zu unflexibel und fehlerhaft ist. Weiterhin können Wartungstätigkeiten an sich zu neuen Wartungsursachen führen.

4 Probleme der Software-Wartung

In diesem Kapitel werden die Probleme beschrieben, die die Wartung erschweren und mit denen sich die Wartungsfachleute auseinandersetzen müssen. Es könnten hier beliebig viele Beispiele aufgezählt werden. Eine vollständige Problembeschreibung für alle Spezialfälle ist nicht möglich. Um die Vielfalt der Probleme besser darstellen zu können, werden diese anhand verschiedener Faktoren eingruppiert. An dieser Stelle wäre auch eine andere Einteilung möglich. Die Kategorien dienen zur Unterstützung der Darstellung der Probleme, und somit halte ich die Aufteilung in wirtschaftliche in 4.1, menschliche in 4.2 und fachliche und inhaltliche Faktoren in 4.3 für gut geeignet. Die wirtschaftlichen Probleme, wie Kosten und Zeit, sind in 4.1.1 und 4.1.2 beschrieben. Menschliche Faktoren sind fehlendes Management in 4.2.1, Software-Dokumentation in 4.2.2, niedriger Stellenwert von Wartungsaufgaben und Einsatz von zweitklassigem Personal in 4.2.3 und Kommunikationsprobleme in 4.2.4. Die fachlichen und inhaltlichen Faktoren, wie schwierige Planbarkeit, Software-Qualität und -Komplexität und das Problem der Werkzeuge, sind in 4.3.1, 4.3.2 und 4.3.3 vorgestellt. Die Untergruppen könnten noch um weitere Beispiele erweitert werden. Die hier erwähnten sollten repräsentativ die Vielfalt der Probleme verdeutlichen.

4.1 Wirtschaftliche Faktoren

Die typischen wirtschaftlichen Probleme sind Kosten und Zeit. In 4.1.1 werden Untersuchungen über die Wartungskosten vorgestellt und danach wird in 4.1.2 der Zeitdruck während der Wartung verdeutlicht.

4.1.1 Kosten

Ursprünglich liegen wirtschaftliche Gründe vor, dass Software-Wartung überhaupt als Problem wahrgenommen wird. Die Betriebsphase eines Software-Systems kann immense Kosten verursachen, die weit über denjenigen der Entwicklung liegen. Beim Management entsteht daher das dringende Anliegen, an dieser Stelle rigoros Kosten einzusparen. Maßnahmen zur Senkung der Wartungskosten sind oft zum Scheitern verurteilt. Im Gegenteil, je länger ein Software-System im Einsatz bleibt, desto größer werden üblicherweise die Wartungskosten [Fäs96: 13].

Die wahrscheinlich erste Schätzung der Wartungskosten stammt von [Can72 zitiert nach [Eis88: 61]]. Nach seinen Untersuchungen ist damals normalerweise etwa 50% des EDV-Budgets für Wartung aufgewendet worden, und in Ausnahmefällen bis zu 80% (diese Schätzung hat übrigens erst die Fachwelt auf die hohen Kosten der Wartung aufmerksam gemacht). Die Hoskyns-Studie in 905 britischen Installationen [Hos73 zitiert nach [Eis88: 61]] ergab, dass fast 40% der Software-Kosten der Wartung zuzurechnen seien. Auch Boehm [Boe76 zitiert nach [Eis88: 61]] kam zum Ergebnis, dass etwa 40% der EDV-Budgets für die Software-Wartung aufgewendet werden. Er nahm allerdings an, dass der Anteil der Wartung im Steigen begriffen sei und schätzte ihn auf 60% für das Jahr 1985. Die Studie von Lientz und Swanson [Lie80 zitiert nach [Eis88: 61]] kommt zum Schluss, dass etwa gleichviel Geld für Neuentwicklung wie für Wartung ausgegeben wird. Die Ursachen für die hohen Kosten lassen sich zum Teil damit begründen, dass sowohl in der Entwicklung als auch in der Wartung zuwenig systematisch vorgegangen wird.

4.1.2 Zeit

Häufig steht man in der Wartung unter großem Termindruck. Das Management drängt auf schnelle und kostengünstige Anpassungen. Unter solchen Voraussetzungen kann der Software-Qualität bei den Wartungsaktivitäten nicht die volle Aufmerksamkeit gewidmet werden (siehe Abb. 4.1). Für die Anwendung strukturierter Wartungsmodelle bleibt häufig keine Zeit [Fäs96: 14].

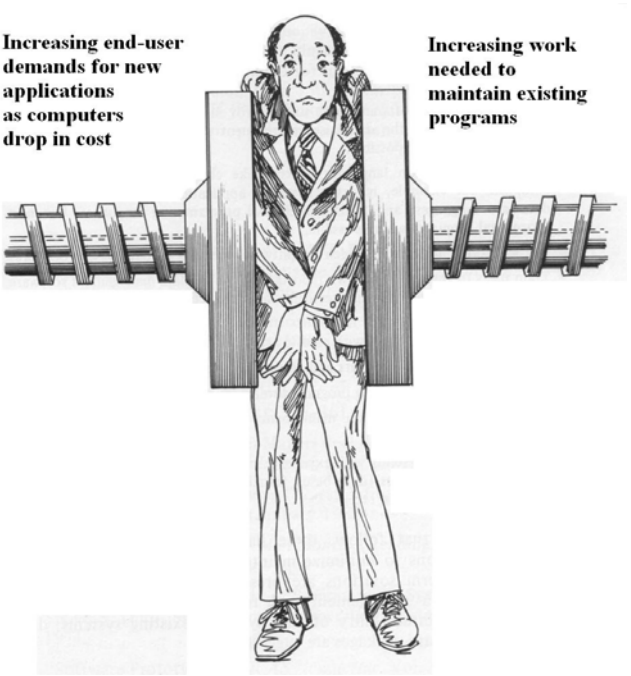


Abbildung 4.1 [Mar83: 13]

4.2 Menschliche Faktoren

Die Wartung ist auch mit Problemen konfrontiert, die von den menschlichen Eigenschaften abhängig sind. Das fehlende Management bei den Wartungstätigkeiten ist ein Problem, für welches das Führungspersonal zuständig ist; es wird in 4.2.1 beschrieben. Ein weiteres immer wieder diskutiertes Problem ist die Dokumentation der Software. Wenn diese nicht richtig geführt wird, ist das kaum auf die technischen Gegebenheiten zurückzuführen. Im Gegenteil, die technische Unterstützung erleichtert oft die Dokumentenerstellung. Aber aus menschlicher Nachlässigkeit wird dieser dennoch nicht genug Aufmerksamkeit gewidmet. Das Problem wird in 4.2.2 geschildert. Danach soll in 4.2.3 darauf hingewiesen werden, dass auch zwischenmenschliche Kommunikation eine Rolle bei der Wartung spielt. Zum Schluss wird in 4.2.4 verdeutlicht, dass bei Mitarbeitern die Wartung einen niedrigen Stellenwert einnimmt und oft nur zweitklassiges Personal eingesetzt wird.

4.2.1 Fehlendes Management

Wie wir gesehen haben, ist der Begriff der Software-Wartung ein sehr allgemeiner Begriff, so allgemein, dass praktisch alle Änderungstätigkeiten an Software als Wartungstätigkeit aufgefasst werden können. Auch die Unterteilung in korrektive, adaptierende, perfektionierende und präventive Wartung lässt noch sehr viel Spielraum für Interpretationen. Es fehlt an einem organisierten Vorgehen in diesen Bereichen [Mül97: 9]. Thomas drückt es so aus:

> „but the problem of software maintenance is, above all, a management problem" (zitiert nach [Hei87: 44]).

Die Bedeutung der organisatorischen Gestaltung des Wartungsprozesses soll das folgende Beispiel von Sneed veranschaulichen (zitiert nach [Bal01: 974]):

„In einer gewöhnlichen Ist-Situation ruft ein betroffener Anwender den zuständigen Entwickler an und fordert ihn auf, einen Fehler zu korrigieren oder eine Änderung durchzuführen. Möglicherweise wird er ihm einen Schmierzettel mit einigen ergänzenden Notizen zukommen lassen.

Der Entwickler nimmt den Antrag an, schaut sich einen Code am Bildschirm an, ändert einige Zeilen, übersetzt das Programm neu und bittet den Anwender, es auszuprobieren. Der Anwender tut dies auch und stellt fest, dass gar nichts mehr läuft. Dann geht die Sucherei los. Das Programm wird noch mehrmals geändert, ehe es zur Zufriedenheit des Anwenders läuft. Inzwischen meldet sich ein anderer Anwender damit, dass seine Ausgaben seit der letzten Änderung nicht mehr stimmen. Jetzt fängt der Kreislauf von vorn an. Der Entwickler hatte vor, die Entwurfsdokumentation noch anzupassen, aber angesichts der neuen Probleme kommt er nicht mehr dazu. Der Code wandert immer mehr vom Entwurf ab. Das Fachkonzept stimmt überhaupt nicht mehr. Es wird nur zu historischen Zwecken aufbewahrt.

Bald ist der Code so oft geflickt worden, dass der Entwickler sein eigenes Gedankengut nicht mehr versteht. Er möchte es gerne überarbeiten, aber dazu hat er keine Zeit. Die Fehlermeldungen und Änderungsanträge häufen sich. Die Anwender melden sich immer ungeduldiger, der Entwickler wird immer frustrierter. Am Ende bleibt ihm nur noch die Kündigung als Ausweg aus der Misere."

Dieses sicherlich überzeichnete Beispiel illustriert einen möglichen Ist-Zustand des Wartungsprozesses. Dies entspricht einer Wartung nach dem Quick-Fix-Modell (Abbildung 4.2). Das Quick-Fix-Modell macht den Programmtext zum zentralen Gegenstand der Wartung. Meistens aus der Notwendigkeit heraus, dass häufig nichts anderes verfügbar ist [Fäs96: 13]. Änderungen werden nur in dem Programmtext durchgeführt, dann wird getestet, recompiliert und das geänderte System wieder eingesetzt [Leh91: 13].

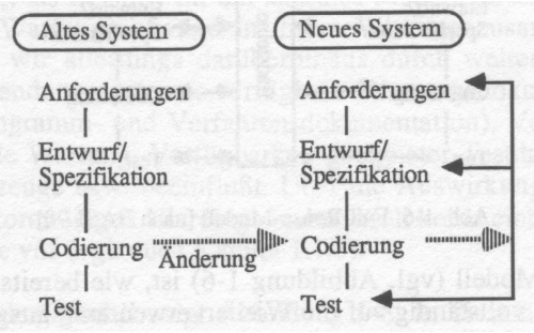


Abbildung 4.2 [Bas90 zitiert nach [Leh91: 13]]

4.2.2 Dokumentation

Wenn schon bei der Entwicklung nicht strukturiert vorgegangen wird, dann sind die Dokumente der Phasen *Anforderungen*, *Analyse* und *Entwurf* praktisch inexistent [Fäs96: 14]. Aber auch wenn noch Dokumente der früheren Entwicklungsphasen vorhanden sind, ist es in der Regel höchst unwahrscheinlich, dass diese nach einer Programmänderung vollständig nachgeführt werden, wie dies auch in dem obigen Szenario von Sneed verdeutlicht ist. Das Software-System und die zugehörige Dokumentation driften immer weiter auseinander. Der gesamte Evolutionsprozess findet häufig nur in der Implementierung statt, wie es auch aus dem Quick-Fix-Modell ersichtlich ist, und ist daher ausschließlich im Programmtext konserviert.

4.2.3 Kommunikation

Es wird immer häufiger beobachtet, dass zwischen Anwender und Wartungspersonal ein Kommunikationsproblem besteht. Bittner und Hesse fanden im Rahmen ihrer Untersuchung heraus, dass die Auslöser für Änderungsanforderungen meist[1] aufgrund der Kundenwünsche entstehen (vgl. [Bit95: 33]). Offensichtlich wird der in den Einzeluntersuchungen beobachtete direktere Zugriff der Fachabteilungen mehr als ausgeglichen durch nachträgliche Wünsche, die aufgrund des geringeren Kontaktes zu externen Kunden entstehen. Bei zu niedriger Benutzerorientierung entstehen Kommunikationsprobleme und es führt zu unzureichendem Systemverständnis sowie zu unrealistischen Erwartungen bei den Anwendern [Leh99: 83].

4.2.4 Niedriger Stellenwert von Wartungsaufgaben und Einsatz von zweitklassigem Personal

Als Ergebnis der ungeplanten und nicht gut organisierten Durchführung der Wartungsaufgaben werden diese in der Praxis oftmals zwecks Einarbeitung und Sammlung von Erfahrungen an weniger qualifizierte Mitarbeiter übergeben, obwohl gerade die Software-Wartung in vielen Fällen aufgrund der vielfältigen Verknüpfungen eine sehr anspruchsvolle Aufgabe darstellt [Hei87: 3]. Trotzdem haben die Wartungsaufgaben unter Fachleuten ein geringes Ansehen [Ei88: 54]. Viele Schwierigkeiten in diesem Gebiet lassen sich direkt oder indirekt darauf zurückführen. Nicht selten werden die Wartungstätigkeiten auch in der

[1] Bis zu 60%

Literatur mit Geringschätzung behandelt. Ein lapidares Beispiel finden wir z.B. in [Bal82 zitiert nach [Ei88: 54]]:

> "Die Software-Entwicklung ist gekennzeichnet durch:
> - einmalige, zeitlich begrenzte Entwicklungsaufgaben
> - interessante, neuartige Probleme,
> - Notwendigkeit schöpferischer, kreativer Ideen.
>
> Die Software-Wartung & Pflege weist folgende Merkmale auf:
> - sich wiederholende, langfristige Tätigkeiten,
> - uninteressante Aufgaben,
> - ereignisgesteuert, d.h. nicht vorhersehbar und daher schwer planbar und kontrollierbar."

Auch Kliewer [Kli99: 36] schreibt über eine negative Einstellung der Entwickler gegenüber der Wartungstätigkeit:

> „Das Wartungsvorhaben an fremder Software gilt in der Praxis allgemein als äußerst schwierig. Meist wird mit dem Hinweis auf die engen Zeitvorgaben und den Termindruck die Neuprogrammierung einer Reengineering-Aktivität vorgezogen. Handelt es sich jedoch um Software, die der Programmierer selbst entwickelt hat, auch wenn dies schon Jahre zurückliegt, sind die Vorbehalte gegen die Wartung und eventuell sogar eine Wiederverwendung von Teilen geringer. Man ist scheinbar nicht bereit, sich in einen fremden Code einzuarbeiten, und sei er von den eigenen Kollegen erstellt. Offensichtlich müssen neben den technischen Problemen, die im Umgang mit älteren Softwaresystemen sicher bestehen, psychologische Barrieren in den Köpfen der Programmierer überwunden werden."

Curth und Giebel schreiben, dass eine immer wieder beobachtete Reaktion der Entwickler, die für Wartungsarbeiten eingesetzt werden, die fehlende Motivation ist. Der Mitarbeiter sieht in der Wartung eine rein reproduktive Aktivität, die seine eigene Kreativität einschränkt. Diese Arbeitsfrustration wird noch dadurch verstärkt, dass die Wartung meist „fremde" Programme betrifft, zu denen er keinen Bezug hat [Cur89: 13].

Es ist schließlich kein Wunder, wenn mit solchen Voraussetzungen und Einstellungen die Entwickler nicht viel Interesse für die Software-Wartung aufbringen und die Wartungstätigkeiten an neue, meist unerfahrene Mitarbeiter verteilt werden.

4.3 Fachliche und inhaltliche Faktoren

Einige Probleme der Wartung sind von fachlicher bzw. inhaltlicher Art. Es ist auf die Software-Eigenschaften selbst zurückzuführen, dass die Planung der Wartung schwer fällt. Wie schwierig die Planbarkeit der Wartung tatsächlich ist, wird in 4.3.1 gezeigt. Danach soll in 4.3.2 erläutert werden, wie sich Software-Qualität und –Komplexität auf die Wartung auswirken. Die Ansicht, dass Qualitäts- und hohe Komplexitätsprobleme durch geeignete Werkzeuge behoben werden könnten, wird widerlegt. In 4.3.3 wird gezeigt, dass von Werkzeugen keine allgemeine Abhilfe zu erwarten ist.

4.3.1 Schwierigkeit der der Wartungsplanung

Das unsystematische Wartungsverhalten entstand sicherlich aus dem Problem heraus, dass eine Systematisierung nur schwer vorzunehmen ist. Die Wartungsaufgaben sind sehr vielseitig und Software hat die Eigenschaft, dass Änderungen am System zu unerwarteten, erschreckenden Seiteneffekten führen können. Durch die Wartung selbst werden immer wieder weitere Fehler verursacht und somit bleiben die Aufgaben kaum überschaubar. Wegen ungeplanter Vorfälle ist die Planung der bevorstehenden Wartungstätigkeiten nur zum Teil möglich. Auch eine Zeit- und Kostenplanung beruht mehr auf Schätzungen als auf Tatsachen. Durch solche Probleme verursachte Systemausfälle können zu Schäden führen, die ungleich größer sind als die Kosten einer „erfolgreichen“ Wartung. Letztendlich ist aber eine genaue Aussage kaum zu treffen, nachdem selbst die Ergebnisse einer erfolgreichen Wartung schlecht messbar sind.

4.3.2 Software-Qualität und Komplexität

Ein weiteres Problem ist auf die Systemgröße und das Alter der Software zurückzuführen. Martin und McClure berichten so darüber [Mar83: 27]:

„Larger and older systems require more maintenance effort than do smaller and younger systems. Software systems tend to grow with age, to become less organized with change, and to became less understandable with staff turnover".

Das in der obigen Aussage erwähnte Problem der Systemgröße ist in der Abbildung 4.3 dargestellt. Logischerweise wächst das System im Entwicklungszeitraum, aber auch nach der Installation ist ein kontinuierliches Wachstum erkennbar.

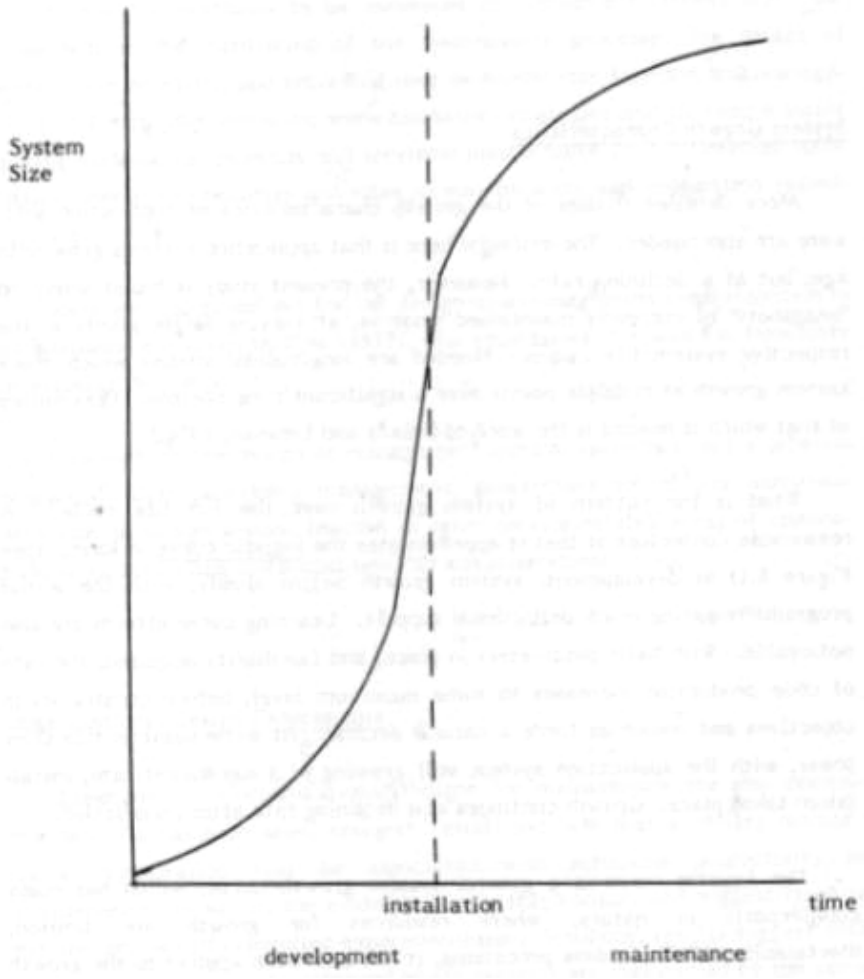


Abbildung 4.3 [Lie80: 164]

Fisher und Standish sagen, dass (die Qualität von) Software durch Wartung verschlechtert wird [Fis79 zitiert nach [Eis88: 54]]. Thomas schreibt ausführlicher [Inf80 zitiert nach [Eis88: 51]]:

„Structure is the opposite of complexity, and complexity is like entropy – it increases in any system unless work is done to decrease it."

Lehman formulierte diesen Satz schon 1974 mit dem Gesetz der wachsenden Komplexität (zitiert nach [web4.1]):

> "As an E-type[2] system evolves its complexity increases unless work is done to maintain or reduce it."

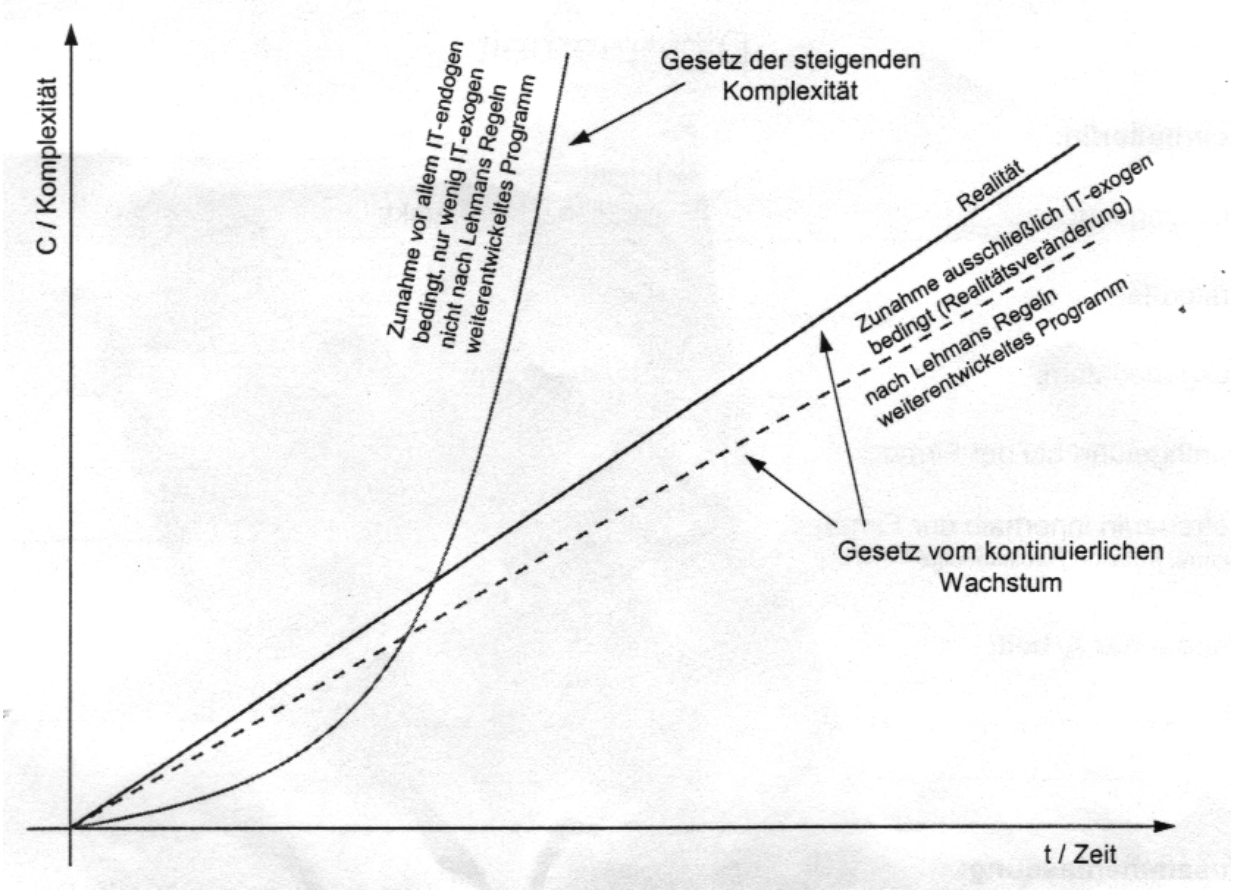


Abbildung 4.4

Abbildung 4.4 verdeutlicht, dass bei einem Programm, das nach Lehmans Regeln weiterentwickelt wurde, die Komplexität entsprechend der Realitätskomplexität wächst. Bei einem Programm, bei dem der Komplexität nicht entgegen gewirkt wurde, steigt diese weit über die Realitätskomplexität hinaus. Die meisten Veränderungen der Software führen dazu, dass die weiteren Änderungen noch schwieriger werden. Man sollte sich also vor einer Modifikation genau überlegen, ob sie wichtig genug ist – auch aus technischer Sicht [Eis88: 51].

[2] E-type Programme werden in 7.2 ausführlich besprochen.

Wallis fasst diese Überlegungen in [Inf80 zitiert nach [Eis88: 51]] folgendermaßen zusammen:

> „The maintenance activity can be characterised as consisting basically in a reconciliation of two aims – the need to accommodate requests for change and the need to control the complexity of software".

Wallis nennt hier zwei konkurrierende Ziele für die Software-Wartung. Einerseits gilt es, das Software-System weiterzuentwickeln und notwendige Änderungen einzubauen, andererseits muss man bei Änderungen zurückhaltend sein, da sie die Struktur und somit die Qualität des Systems verschlechtern.

Sneed schreibt, dass eine Studie im Rahmen des ESPRIT-METKIT-Projektes gezeigt hat, dass sich die Komplexität eines unstrukturierten Pogramms nach einer Korrektur um 4%, nach einer Änderung um 17% und nach einer Erweiterung um 2% erhöht [Sne92: 20]. Die zunehmende Unüberschaubarkeit eines Programms verursacht einen immer höheren Wartungsaufwand, was als logische Folge hat, dass Wartungsaufgaben generell immer teurer werden.

4.3.3 Werkzeuge

Das am Markt verfügbare Angebot an Werkzeugen ist relativ unübersichtlich und es existiert bis heute weder ein allgemein anerkanntes Klassifikationsschema, das bei der Einsatzplanung helfen könnte, noch ein einigermaßen vollständiger Produktkatalog [Leh99: 89].
Martin und McClure berichten, dass es bei den Werkzeugen einige Probleme gibt, obwohl am Markt eine Vielzahl von ihnen zur Verfügung steht. Es mangelt ihnen an Einheitlichkeit, Vollständigkeit und Kompatibilität. Es gibt kein IT-System, das über eine vollständige Werkzeugausstattung verfügen würde. Viele der angebotenen Werkzeuge sind von dem System und der Sprache abhängig. Sogar Werkzeuge für das gleiche System und für die gleiche Sprache können inkompatibel sein [Mar83: 405].
McClure belegt anhand von Studien, dass trotz der Entwicklung komfortabler und mächtiger Werkzeuge der Software-Wartungsaufwand nicht reduziert, sondern bestenfalls begrenzt werden konnte [Clu93 zitiert nach [Kli99: 18]].

5 Ziele der Software-Wartung

Die Ziele der Wartung können anhand der Definitionen, der Ursachen und der Probleme der Wartung abgeleitet werden. Zuerst sollen in 5.1 die Ziele entsprechend der Definition der Wartung aus 2.2 dargestellt werden. Danach erfolgt in 5.2 eine Zielbeschreibung anhand der in 3.2.2 erwähnten Wartungsursachen. Hier wird auch eine Einteilung nach technischen in 5.2.1 und konzeptionellen Aspekten in 5.2.2 vorgenommen. Zum Schluss wird in 5.3 gezeigt, dass sich die Ziele auch aus den Problemen der Software-Wartung ergeben. Diese lassen sich, ähnlich wie in Kapitel 4 anhand wirtschaftlicher in 5.3.1, menschlicher in 5.3.2 sowie fachlicher und inhaltlicher Kriterien in 5.3.3 beschreiben.

5.1 Ziele anhand der Definition der Software-Wartung (Bezug zu 2.2)

Die Software-Wartung wurde in Kapitel 2 definiert. Hier sollen die Ziele besonders in Bezug zu 2.2 beschrieben werden. Dort wurde die Wartung im Hinblick auf verschiedene Tätigkeitsbereiche definiert. Bei der Zielbeschreibung kann eine parallele Einteilung erfolgen. Curth und Giebel haben die Ziele gemäß ihrer Definitionen der Wartung festgelegt. Demnach können die Ziele ebenfalls in korrektive, adaptive und perfektive Ziele aufgeteilt werden [Cur89: 80]:

> „Korrektive Ziele: Aufdeckung und Korrektur von Softwarefehlern, Behebung von Performance-Engpässen und Korrektur von Implementierungsfehlern.
>
> Adaptive Ziele: Anpassung der Software an Änderungen innerhalb der Datenanforderungen oder der Systemumgebung.
>
> Perfektive Ziele: Erhöhung der Performance, Steigerung der Softwarerentabilität sowie Verbesserung des Rechnerdurchsatzes und der Wartbarkeit."

5.2 Ziele anhand der Kategorien der Wartungsursachen (Bezug zu 3.2.2)

In Kapitel 3 wurden die Ursachen der Wartung vorgestellt. Hier sollen die Ziele besonders in Bezug zu 3.2.2 beschrieben werden. Dort sind die Ursachen nach technischen sowie nach konzeptionellen Aspekten aufgezählt (vgl. 3.2.2.1 bzw. 3.2.2.2). Eine entsprechende

Einteilung ist auch bei der Zieldefinition möglich. Die Ziele der Wartung werden aus technischer Sicht in 5.2.1 und aus konzeptioneller Sicht in 5.2.2 aufgelistet.

5.2.1 Technische Ziele (Bezug zu 3.2.2.1)

Aus den Wartungsgründen resultieren aus technischer Sicht die folgenden Ziele: Hardware- und Software-Komponenten nach den aktuellen Bedürfnissen anzupassen, Softwarefehler aufzudecken und zu beseitigen. Auf längere Sicht sollten die Programme überarbeitet und eventuell neu strukturiert werden, um eine bessere Wartbarkeit zu erzielen sowie eine mögliche Wiederverwendung vorzubereiten.

5.2.2 Konzeptionelle Ziele (Bezug zu 3.2.2.2)

Die konzeptionellen Gründe für Wartung (vgl. 3.2.2.2) wurden nach internen bzw. externen Aspekten aufgeteilt. Zusammenfassend handelt es sich dabei um Veränderungen der Umwelt. Ziel der Wartung ist es, auf Veränderungen innerhalb bzw. außerhalb des Unternehmens rechtzeitig und effizient zu reagieren. Bei Veränderungen der Systemumgebung, der Unternehmensorganisation, der Geschäftsprozesse und der Benutzeranforderungen (vgl. 3.2.2.2.1) soll eine Software-Anpassung erfolgen. Auch externe Faktoren haben eine Auswirkung auf die Software (vgl. 3.2.2.2.2): Wenn sich Rahmenbedingungen, das Verhalten der Konkurrenz, die Gesetze und der Markt verändern, so soll auch die Software in diesem Prozess inbegriffen sein. Eine Software-Alterung ist nur dann zu vermeiden, wenn diese mit den Veränderungen der Umwelt Schritt halten kann. Sonst verliert die Software ihre Aktualität und kann ihren Zweck nicht mehr erfüllen. Ziel der Wartung ist es, dem Alterungs-Prozess entgegenzuwirken und die Lebensdauer der Software zu verlängern.

5.3 Ziele anhand der Probleme der Software-Wartung (Bezug zu 4)

In Kapitel 4 wurden die Probleme der Wartung aufgezählt. Die Vielfalt der Probleme spiegelt sich auch in der Zielfindung wider. Dabei können wirtschaftliche in 5.3.1, menschliche in 5.3.2 sowie fachliche und inhaltliche Teilziele in 5.3.3 festgelegt werden.

5.3.1 Ziele nach wirtschaftlichen Faktoren (Bezug zu 4.1)

Wirtschaftliche Ziele sind, das Kostenproblem (vgl. 4.1.1) zu lösen und den Zeitdruck (vgl. 4.1.2) zu mindern. Bezüglich der Kosten formuliert Lehner folgende Zielsetzung [Leh91: 47]:

> „Zweck der Wartung ist es, die Investitionen des Unternehmens in Anwendungssysteme zu schützen, indem deren sinnvolle Nutzungsdauer verlängert und deren Nutzen für das Unternehmen verbessert werden."

5.3.2 Ziele nach menschlichen Faktoren (Bezug zu 4.2)

Menschliches Ziel ist es, die Wartungsfaktoren zu verbessern, die vom Personal abhängig sind. Die Führungsebene soll Kompetenz darin erlangen, Managementaufgaben wahrzunehmen und Unterstützung bei deren Durchführung zu bieten [Leh99: 83] (vgl. 4.2.1). Viele der Probleme, die auf menschliche Aspekte zurückzuführen sind, können durch eine geeignete Managementführung gemindert werden. Somit kann die Dokumentenführung (vgl. 4.2.2), wie auch die Kommunikation zwischen Anwender und Wartungspersonal (vgl. 4.2.3) durch das Management gefördert werden. Kommunikationsprobleme sollten durch die Verbesserung des Informationsflusses zwischen den Beteiligten gelöst werden. Bezüglich der Dokumente ist ein Optimum zu erreichen, d.h. eine aktuelle bzw. vollständige Dokumentenführung. Kliewer weist darauf hin, dass eine Dokumentation in übersichtlich formatierten und/oder graphischen Darstellungen die Einarbeitung erheblich erleichtert [Kli99: 22]. Das Management hat auch dafür zu sorgen, dass für die Wartung qualifizierte Mitarbeiter eingesetzt werden (vgl. 4.2.4). Hierbei ist das Ziel, die Aufgaben für die Mitarbeiter positiver darzustellen, damit die negative Einstellung gegenüber der Wartung verbessert werden kann.

5.3.3 Ziele nach fachlichen und inhaltlichen Faktoren (Bezug zu 4.3)

Fachliche und inhaltliche Ziele ergeben sich anhand der Wartungsgründe bezüglich der Planbarkeit der Wartung (vgl. 4.3.1), der Software-Qualität bzw. Komplexität (vgl. 4.3.2) und der Werkzeuge (vgl. 4.3.3). Eine besser planbare Wartung hängt eng mit der Qualität und Komplexität der Software zusammen. Um das System überschaubar zu halten, ist es Ziel der

Wartung, gegen die Komplexität und für eine bessere Software-Qualität aufzutreten. Dann wird die Planbarkeit der Wartungstätigkeiten auch vereinfacht. Das Werkzeugangebot kann die Wartung selbst nicht verbessern, aber es kann zum Ziel gesetzt werden, die am Markt vorhandenen Werkzeuge besser einzusetzen. Bittner und Hesse weisen darauf hin [Bit85: 81]:

> „Wünscht man sich eine konstruktive Mitarbeit der Entwickler bei der Ausstattung ihres Arbeitsplatzes, so setzt dies voraus, dass man ihnen vorher die Möglichkeiten gibt, die in Frage kommenden Werkzeuge kennen zu lernen."

6 Vorgehensmodelle der Software-Wartung

In diesem Abschnitt werden verschiedene Wartungsstrategien untersucht. Wartung kann auf verschiedenen hohen Abstraktionsebenen erfolgen. In 6.1 wird die Wartungspyramide vorgestellt, die eine Übersicht der verschiedenen Abstraktionsniveaus bietet. Die Pyramide beinhaltet auch den Reengineering-Prozess. In diesem Zusammenhang wird in 6.2 die Software-Reengineering besprochen. Zunächst soll in 6.2.1 Reengineering definiert und danach in 6.2.2 auch die mit Reengineering in Verbindung gebrachten Begriffe erläutert werden. In 6.3 wird auf die einzelnen Abstraktionsniveaus, die in 6.1 und 6.2.2 angesprochen wurden, näher eingegangen. In 6.3 wird der Wartungsansatz mit niedrigstem Abstraktionsniveau besprochen. Dazu soll in 6.3.1 die Wartung auf der Codierungsebene beschrieben werden. Danach folgt in 6.3.2 ein Restrukturierungsprozess von Sneed, der ebenfalls auf der niedrigsten Ebene verläuft. In 6.4 werden Wartungsansätze mit mittlerem Abstraktionsniveau vorgestellt. In 6.4.1 wird zuerst eine Methode von Waters, Transformation nach Abstraktion und Reimplementierung, gezeigt und danach folgt in 6.4.2 ein Bespiel für Redesign von Sneed. In 6.4.3 ist ein Restrukturierungsprozess von Kaufmann zu finden. Dieser wird ausführlich in 6.4.3.1, 6.4.3.2 bzw. 6.4.3.3 erklärt. Eine Stellungnahme zu dem Prozess wird in 6.4.3.4 geboten. Anschließend werden in 6.4.4 die Erkenntnisse und die wichtigsten Punkte zu der Wartung auf dem mittleren Abstraktionsniveau zusammengefasst. Die Wartungsansätze mit höherem Abstraktionsniveau werden in 6.5 vorgestellt. Die konzeptionellen Aspekte können mit Hilfe der Modellierung erfasst werden. In 6.5.1 soll ein Überblick der Modelle für die Software-Wartung geboten werden. Danach ist in 6.5.2 ein Beispiel für ein Vorgehensmodell der Software-Wartung nach Curth und Giebel zu finden. In 6.5.3 ist das Thema Evolution während der Software-Entwicklung und –Wartung. Ein umfassendes Evolutions-Modell für den gesamten Lebenszyklus eines Systems wird in 6.5.3.1 vorgestellt. Danach werden die Zeiträume auch einzeln besprochen: In 6.5.3.2 wird die Evolution während der Software-Entwicklung diskutiert und in 6.5.3.3 folgt die Software-Evolution, welche während des Zeitraums der Wartung stattfindet. Dazu wird zuerst in 6.5.3.3.1 der Begriff „Software-Evolution“ erläutert. Danach wird auf die Idee von Lehman zurückgegriffen. Die S-, P- und E-Programmtypen bilden die Basis für die Evolutionslehre im Bereich der Software, welche in 6.5.3.3.2 zu finden sind. Danach wird in 6.5.3.3.3 der Software-Evolutionsansatz von Lehman weiter ausgeführt.

6.1 Software-Wartungspyramide

Wartungstätigkeiten können auf verschiedenen Abstraktionsebenen durchgeführt werden. Abbildung 6.1 zeigt die Wartungspyramide mit den verschiedenen Ebenen. Innerhalb der Wartungspyramide befindet sich die Software-Reengineering-Pyramide (vgl. 6.2.2), welche nur die technische Ebene der großen Pyramide abdeckt (siehe Abbildung 6.1):

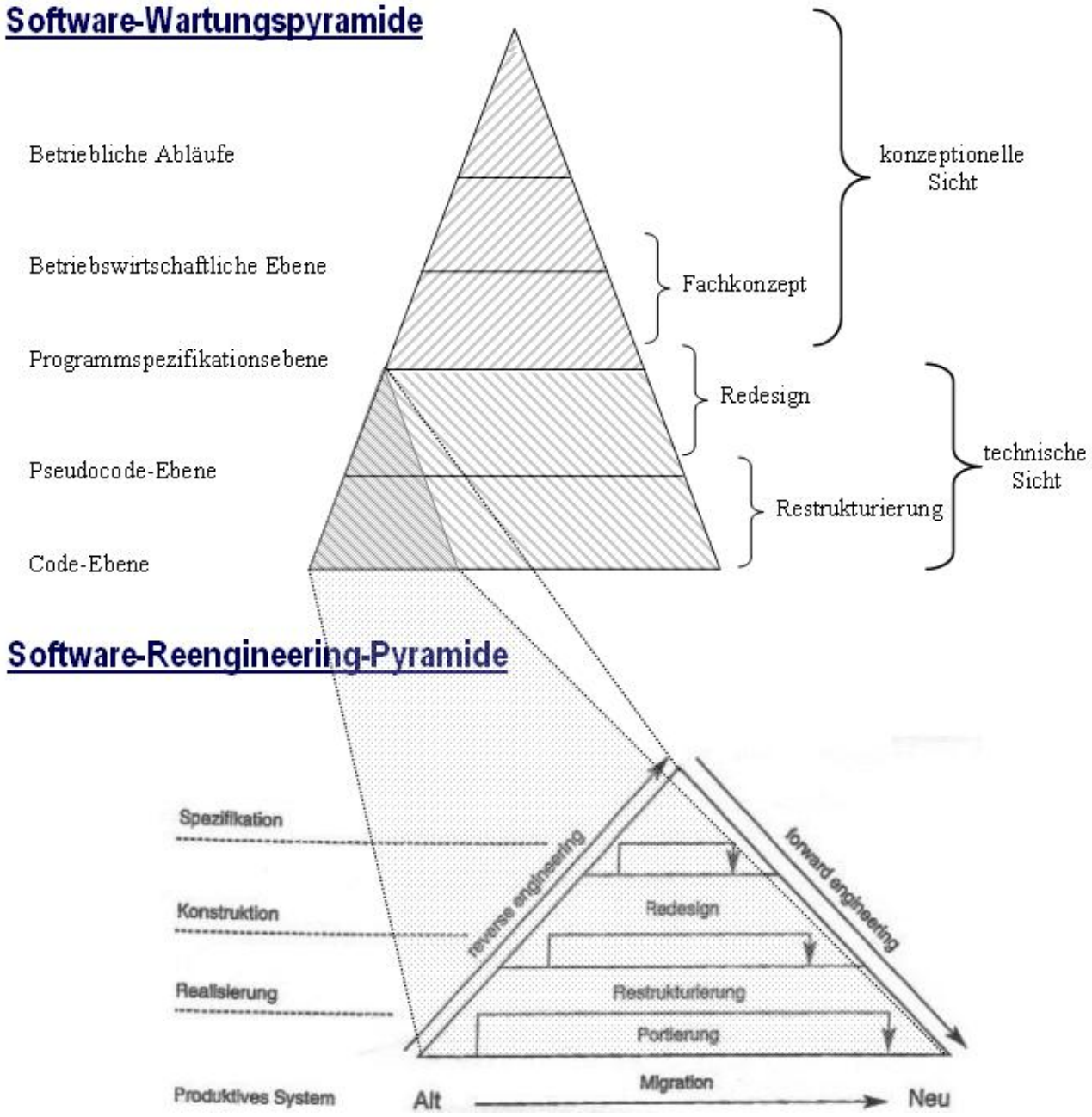


Abbildung 6.1

In der Wartungspyramide können drei verschiedene Abstraktionsniveaus definiert werden: das niedrigste, das mittlere und das höhere Abstraktionsniveau. Die zwei unteren Niveaus

sind auf der technischen Ebene. Nur das höhere Niveau erfasst auch die konzeptionelle Sicht. Hier soll eine Übersicht der Ebenen geboten werden. In den folgenden Kapiteln wird auf die einzelnen Bereiche näher eingegangen.

Die technische Ebene:

- Das niedrigste Abstraktionsniveau befasst sich nur mit dem Programmieren. Das Wartungspersonal kann sich von der Programmiersprache lösen und durch die Verwendung eines Pseudocodes Restrukturierungsmaßnahmen durchführen. Werden die Wartungstätigkeiten nur auf dieser Ebene ausgeführt, so kann man von einem unvollständigen Reengineering sprechen (vgl. 6.2 und 6.3).

- Auf dem mittleren Abstraktionsniveau wird das IT-Konzept erforscht. Anhand des Codes wird das Design des Systems wieder gewonnen. Es geschieht auf der Programmspezifikationsebene (vgl. 6.2 und 6.4). Hier kann der Entwurf überarbeitet und ein Redesign durchgeführt werden. Wenn die Programmspezifikationsebene erreicht wird, kann von einem vollständigen Reengineering die Rede sein (vgl. Abb. 6.1).

Die konzeptionelle Ebene:

- Auf den höheren Abstraktionsniveaus werden erst die konzeptionellen Aspekte des Einsatzbereiches berücksichtigt. Das Fachkonzept beinhaltet das Modell des Betriebs (deskriptives Modell) und das Modell der Software (präskriptives Modell).

6.2 Software-Reengineering

Wie in 6.1 gezeigt wurde, kann Wartung auf unterschiedlich hohem Abstraktionsniveau erfolgen. Wenn die Wartungstätigkeiten auf den untersten bis mittleren Ebenen ausgeführt werden, sind diese nur auf die technischen Aspekte begrenzt (vgl. 2.1.1, 3.2.2.1 und 5.2.1). In Abbildung 6.1 entspricht die technische Sicht einem Reengineering-Prozess, auf den hier näher eingegangen werden soll. Zuerst wird in 6.2.1 „Reengineering“ anhand von Zitaten definiert. Danach soll in 6.2.2 die Reengineering-Pyramide ausführlich besprochen und somit auch die Begriffe, die mit Reengineering zusammenhängen, erklärt werden.

6.2.1 Definitionen

In der Literatur sind zahlreiche Begriffserklärungen von „Reengineering“ zu finden. Einige möchte ich an dieser Stelle zitieren:

> „Software-Reengineering im weiteren Sinne bedeutet die Nutzung bestehender Altsysteme (Programmcode und Dokumentation) als Informationsquelle zur Wartung und Neuentwicklung von Anwendungen“ [Kau96 zitiert nach [Kli99: 3]].

> „Mit Reengineering ist Programm-Sanierung gemeint. Alte Programme werden restrukturiert und renoviert mit dem Ziel, ihre Qualität zu steigern. Durch die verbesserte Qualität wird der Wartungsaufwand gesenkt und die Wiederverwendbarkeit der Software erhöht. Reengineering ist oft eine Voraussetzung für Reverse-Engineering, denn nur strukturierte und modulare Programme lassen sich nachmodellieren“ [Sne92: 12].

> „Software Reengineering ist die praktische Anwendung wissenschaftlicher Erkenntnisse einschließlich der verfügbaren Methoden, Techniken und Werkzeuge für Wartung und Pflege, Weiterentwicklung, Portierung bzw. Migration und systematisches Ablösen von Softwaresystemen, die sich im Einsatz befinden, unter Berücksichtigung wirtschaftlicher Anforderungen. Dies umfasst auch die Softwarewartung, welche zum Software Reengineering eine analoge Stellung einnimmt, wie die Programmierung zum Software Engineering“ [Leh99: 85].

Die oben aufgeführten Definitionen mögen unterschiedlich erscheinen. Deswegen wird der Begriff „Reengineering“ auf einem anderen Weg verdeutlicht: Software Engineering ist das ingenieurmäßige Vorgehen während der Software-Entwicklung. Software-Reengineering setzt dagegen voraus, dass ein System bereits entworfen ist. Der Begriff beschreibt somit das ingenieurmäßige Vorgehen, das an einem fertigen, im Einsatz befindlichen System durchgeführt wird. Die einzelnen Tätigkeitsbereiche, welche die obigen Definitionen zu beschreiben versuchen, sind dabei recht vielfältig. Eine ausführlichere Erklärung dazu folgt in 6.2.2.

6.2.2 Reengneering-Pyramide

Software-Reengineering kann auf mehrere Teilbereiche aufgeteilt werden, auf die ich im Folgenden näher eingehen möchte. Um zum besseren Verständnis des „Reengineerings“ beizutragen, werde ich nach Darstellung der Teilbereiche die darin enthaltenen wichtigen Begriffe erklären.

Siedersleben [Sie03: 166] bezieht sich auf [JaL91], bei dem Reengineering der Prozess der Analyse des Altsystems ist, die Festlegung von Änderungen auf der Design-Ebene und die Reimplementierung. Dies wird ausgedrückt in der Formel:

Reengineering = Reverse-Engineering + Δ + Forward Engineering (Abb.6.2)

Δ steht für die Modifikationen – sie ergeben sich aufgrund von Änderungen der Funktion, des Designs oder der Implementierungstechnik. Anstelle von Reengineering verwendet Siedersleben den Begriff Software-Renovierung [Sie03: 166].

Die Besonderheit der Software-Renovierung – im Unterschied zu Software-Wartung im gängigen Software-Entwicklungsprozess – ist die zusätzliche Phase des Reverse-Engineering: Sie schafft die notwendige Grundlage für die Weiterentwicklung. Deshalb ist Software-Renovierung - laut [Sie03: 166] - aufwendiger als Software-Wartung. Software-Renovierung geschieht auf verschiedenen Stufen: Spezifikation, Konstruktion und Realisierung; dabei werden verschiedene Projekttypen eingesetzt: Redesign, Restrukturierung, oder Portierung.

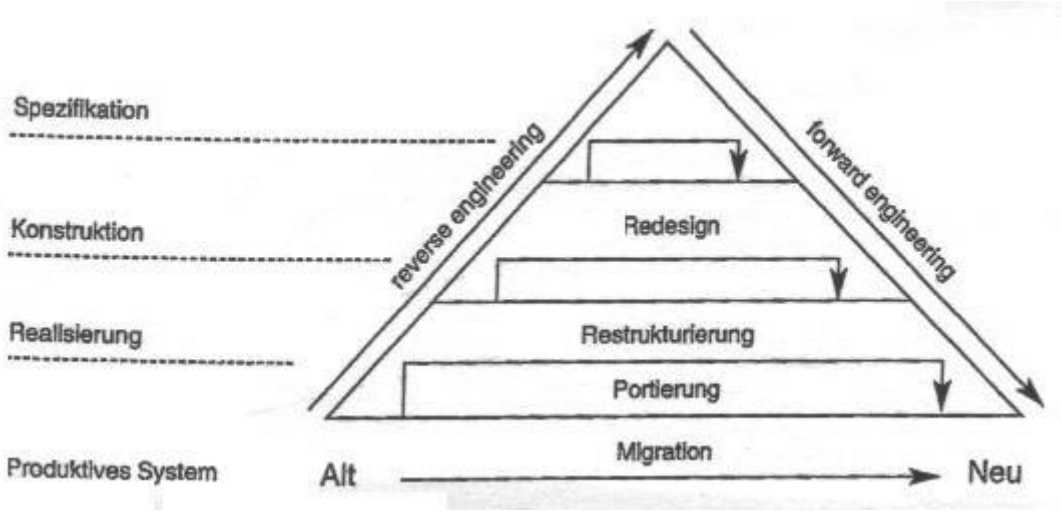


Abbildung 6.2 [Sie03: 167]

Hierzu einige Begriffserklärungen (vgl. [Sie03: 166]):

Reverse-Engineering bezeichnet den Analyse-Prozess, dem ein bestehendes Software-System unterzogen wird, um aus der Realisierungsebene Informationen höherer Ebenen (Design und Spezifikation) zu gewinnen. Notwendig ist das immer dann, wenn eine Dokumentation nicht vorhanden ist, nicht mehr mit der Realisierung übereinstimmt oder nicht ausreichend genau ist. Dabei will man die Struktur und die Zusammenhänge des Systems erkennen und zweckmäßig darstellen, wie z.B. in Daten- und Kontrollflussdiagrammen oder in Entity-Relationship-Diagrammen.

Forward Engineering ist der traditionelle Software-Entwicklungsprozess; ausgehend von den Anforderungen der Anwender führt er über die implementierungsunabhängige Spezifikation zu der Konstruktion und der technischen Realisierung.

Migration stellt den Übergang zwischen Alt- und Neusystem dar.

Redesign bezweckt die Erhöhung der technischen Qualität eines ganzen Programmsystems, es beginnt auf der Konstruktionsebene. Dort werden die Konstruktionsinformationen des Altsystems durch Reverse-Engineering zurückgewonnen, dann erfolgt die Festlegung eines neuen Designs und nachfolgend werden im Forward Engineering die geänderten Systemteile neu realisiert.

Restrukturierung soll die technische Qualität innerhalb der Module erhöhen und wird auf der Realisierungsebene durchgeführt. Der Anteil des Reverse-Engineering ist geringer als beim Redesign und damit die Kosten niedriger. Erreicht wird die Verbesserung der Programm- oder Datenstruktur durch eine Vereinheitlichung von Parameterübergabe und Fehlerbehandlung, durch Ersetzen von direkten Sprungbefehlen oder durch Entfernen von unnötigen Datenzugriffen und redundanten Codes. Auch die fachliche und technische Nachdokumentation sind wichtige Maßnahmen der Restrukturierung.

Portierung hat einen Wechsel der Systemplattform oder der Sprache zum Ziel. Sie kostet weniger; Reverse- und Forward-Engineering beschränken sich auf die niedrigste Ebene, die Realisierung. Durch den hohen Formalisierungsgrad sind die Umsetzungen weitgehend automatisierbar; strukturelle Verbesserungen erreicht man aber nicht.

6.3 Wartungsansätze mit niedrigstem Abstraktionsniveau

Hier sollen die zwei Ebenen besprochen werden, die sowohl in der Wartungspyramide (Abb. 6.1) wie auch in der Reengineering-Pyramide (Abb. 6.2) zu sehen waren. In 6.3.1 wird die Wartung auf der Code-Ebene beschrieben. Danach wird in 6.3.2 ein Beispiel von Sneed vorgestellt, das ebenfalls auf die niedrigste Ebene beschränkt ist.

6.3.1 Code-Ebene

Eine bis jetzt oft angewendete Möglichkeit ist, Wartung als Programmierung zu verstehen. Die Beschränkung auf die Codierungsebene führt aber nicht zu dem erwünschten Ergebnis und entspricht der technischen Sicht in 2.1.1 bzw. den technischen Aspekten in 3.2.2.1, sowie den technischen Zielen in 5.2.1, ohne die konzeptionellen Aspekte beachtet zu haben.

Auf dem Gebiet der Programmierung wurden bislang die meisten Erfahrungen gesammelt. Es gibt nicht nur eine reichhaltige Literatur zu diesem Thema, sondern auch einige praktische Hilfsmittel. Für die Restrukturierung unstrukturierten Codes wurden sogar Expertensysteme angeboten, über deren wahren Nutzen sich allerdings streiten lässt [Sne92: 58]. Trotzdem gibt es eine breite Produktpalette, Super Structure, Structured Engine und Recorder sind Beispiele solcher Restrukturierungsautomaten [Dat86 zitiert nach [Sne92: 58]].

Die bisherigen langjährigen Programmiererfahrungen sowie die am Markt angebotenen Werkzeuge führten zu dem Ergebnis, dass Wartung bisher hauptsächlich auf der Codierungsebene stattgefunden hat. Dies entspricht dem Quick-Fix-Modell, das in 4.2.1 beschrieben worden ist. Oft wird dabei eine Zwischensprache benutzt. In einem Artikel in den IEEE Transactions on Software Engineering beschreibt Waters von MIT zwei Möglichkeiten, ein Programm zu transformieren [Wat88 zitiert nach [Sne92: 138]], von denen die erste einem niedrigen Abstraktionsniveau entspricht: Transformation nach Transliteration und Verfeinerung. Diese Vorgehensweise soll hier vorgestellt werden. Die zweite von Waters vorgeschlagene Möglichkeit, Transformation nach Abstraktion und Reimplementierung, geschieht auf dem mittleren Abstraktionsniveau und wird dementsprechend in 6.4.1 besprochen. Die erste Methode, die Transformation nach Transliteration und Verfeinerung (vgl. [Sne92: 138]), wird auf der Code-Ebene durchgeführt. Das Programm wird Anweisung für Anweisung nach Regeln für jeden Anweisungstyp in einen Zwischencode umgesetzt.

Anschließend wird der Zwischencode verfeinert bzw. optimiert. Zum Schluss werden aus dem Zwischencode Anweisungen der Zielsprache erzeugt (siehe Abbildung 6.3).

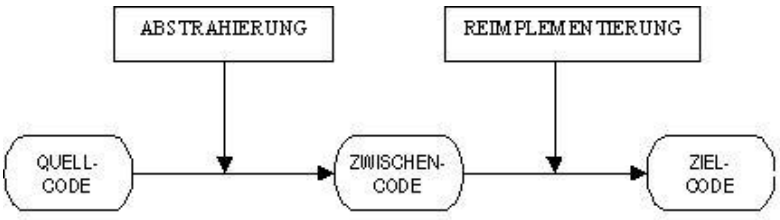


Abbildung 6.3 [Sne92: 139]

McClure fügt hinzu [Mül97: 76], dass die anweisungslokale Konversion prinzipiell keine große Verbesserung[3] des Programm-Codes erzielen kann, da die Überführung Anweisung zu Anweisung die Verwendung ausdrucksstarker Sprachkonstrukte, die die Zielsprache, nicht aber die Quellsprache besitzt, nicht zulässt.

In dieser Diplomarbeit wird betont, dass Wartung auf niedrigem Abstraktionsniveau durchzuführen, zu den veralteten Methoden zählt, die nicht zu dem erwünschten Ergebnis führt.

6.3.2 Restrukturierungsprozess von Sneed

Sneed beschreibt ein Beispiel [Sne92: 146], indem Cobol-74-Programme in Cobol-85-Programme umstrukturiert werden. Er benannt sein Beispiel als ein Reengineering-Prozess, der in fünf aufeinander folgende Schritte abläuft (siehe Abb. 6.4):

- Schritt 1: Statische Analyse des alten Programms und Erstellung einer maschinell gespeicherten Programmdokumentation.
- Schritt 2: Modularisierung des Programms anhand der Programmdokumentation und der Modularisierungskriterien.
- Schritt 3: Strukturierung der neugeschaffenen Module im Pseudo-Code.
- Schritt 4: Optimierung und Anpassung der strukturierten Pseudo-Code-Module
- Schritt 5: Generierung der strukturierten COBOL 85-Module, die in ihrer Gesamtheit die Funktion des alten Programms erfüllen.

[3] Es ist im Allgemeinen sogar eher das Gegenteil der Fall.

Die Schritte 1, 2, 3 und 5 sind voll automatisiert. Schritt 4 verlangt den manuellen Eingriff des Wartungsprogrammierers. Er kann den regenerierten Pseudo-Code-Entwurf mit Hilfe eines syntaxgetriebenen Editors korrigieren.

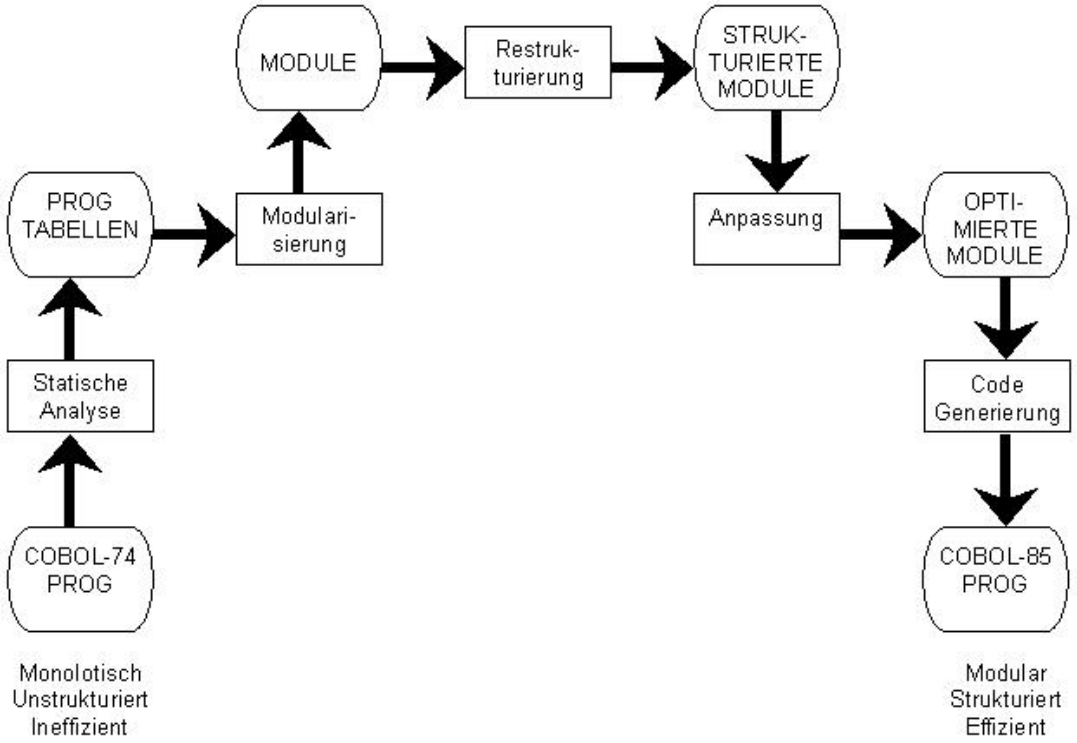


Abbildung 6.4 [Sne92: 149]

Sneed bezeichnet diese Restrukturierung als Reengineering-Prozess. Ein Vergleich mit Abbildung 6.1 und 6.2 verdeutlicht aber, dass die Restrukturierung nur als unvollständiges Reengineering verstanden werden kann: Die Programmspezifiktionsebene wird nicht erreicht. Es kann hier von dem niedrigsten Abstraktionsniveau die Rede sein, bei dem es sich hauptsächlich um das Programmieren handelt. Der erwähnte Pseudocode kann sich zwar von spezifischen Gegebenheiten der Programmiersprache lösen, aber es soll betont werden, dass dabei noch immer auf einer sehr niedrigen Ebene gearbeitet wird. Diesem Prozess entspricht nicht das in 6.2 definierte Reengineering.

6.4 Wartungsansätze mit mittlerem Abstraktionsniveau

Das mittlere Abstraktionsniveau ist in 6.1 und 6.2 dargestellt. Es handelt sich dabei um einen vollständigen Reengineering-Prozess, der auf der Programmspezifikationsebene verläuft. Es soll aber betont werden, dass die Wartungstätigkeiten die konzeptionelle Ebene nicht

erreichen und somit diese nur auf die technischen Aspekte begrenzt (vgl. 2.1.1, 3.2.2.1 und 5.2.1) sind. In 6.3.1 wurde schon die Methode „Transformation nach Transliteration und Verfeinerung“ besprochen. Die zweite Methode von Waters, die Transformation nach Abstraktion und Reimplementierung, soll in 6.4.1 vorgestellt werden. Danach folgen in 6.4.2 ein Prozess von Sneed und ein weiterer in 6.4.3 von Kaufmann. Der letztere kann in mehrere Teilbereiche gegliedert werden, welche in 6.4.3.1, 6.4.3.2 bzw. 6.4.3.3 aufgeführt sind. Eine Stellungnahme dazu erfolgt in 6.4.3.4. Zum Abschluss werden in 6.4.4 die Erkenntnisse und die wichtigsten Punkte zu der Wartung auf dem mittleren Abstraktionsniveau zusammengefasst.

6.4.1 Transformation nach Abstraktion und Reimplementierung nach Waters

Waters schlägt die Transformation nach Abstraktion und Reimplementierung (zitiert nach [Sne92: 138]) vor. Diese ist ein dreistufiges Verfahren, wobei das Programm zunächst in eine völlig andere, semantisch höhere Darstellungsform transformiert wird (siehe Abb. 6.5). Die drei Stufen werden, wie folgt beschrieben:

- Die erste Stufe – die Rekonstruierung der alten Spezifikation (IT-Konzept) – ist das eigentliche Reverse-Engineering. Dabei wird eine globale Analyse des Ausgangssystems vorgenommen, die zu einer abstrakten Beschreibung (Modell) des Programminhalts – unabhängig von den syntaktischen Gegebenheiten von Ziel- und Ausgangssystem – führt.

- In der zweiten Stufe wird diese Abstraktion manipuliert bzw. optimiert. Hier werden die Programm- und Datenstrukturen auf Grund vorgegebener Ziele überarbeitet, z.B. ein Netz in einem Baum umgewandelt, Ablaufgraphen in Teilgraphen zerlegt und Datenflüsse zu Objekten gelegt.

- Die dritte Stufe bildet die Generierung eines neuen Codes aus der abstrakten Beschreibung heraus. Wäre dieses abstrakte Modell nicht manipuliert worden, hätte man jetzt wieder ein Programm von gleicher Funktionalität wie das alte [Bal81 zitiert nach [Sne92: 146]]. Durch die Änderung ergibt sich jedoch eine neue Variante des alten Programms. An diesem Punkt ist es nun möglich, das Programm in einer anderen Programmiersprache neu zu generieren.

Durch die mittelbare Transformation werden die Nachteile der direkten Umsetzung vermieden und in Bezug auf das Zielsystem wird eine optimale Restrukturierung ermöglicht.

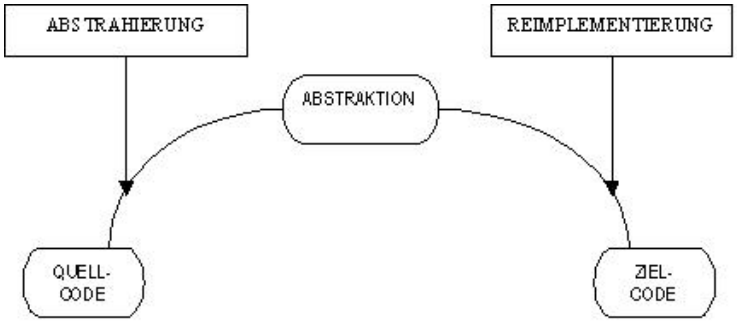


Abbildung 6.5 [Sne92: 140]

Dieses mittelbare Restrukturierungsverfahren ist weitgehend identisch mit einem Reengineering-Prozess. Dabei wird von der Anwendungssystemebene ausgegangen. Durch Reverse-Engineering erreicht man die Implementierungsebene. Hier erfolgt ein Redesign und anschließend führt eine Implementierung (Forward-Engineering) wieder zurück zur Ausgangsebene [Kau94: 33] (Abb. 6.6).

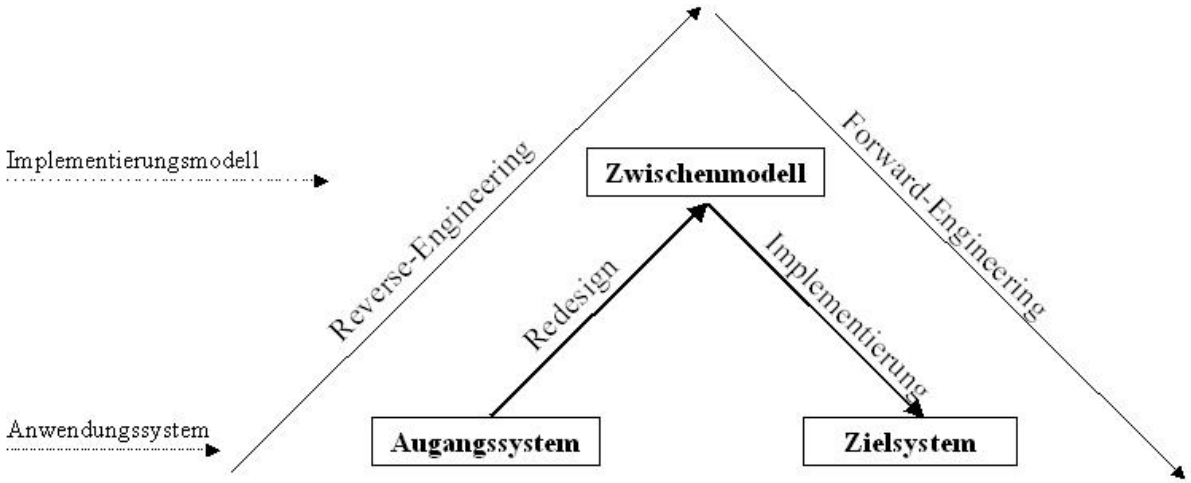


Abbildung 6.6 [Kau94: 34]

6.4.2 Redesign nach Sneed

Auch Sneed beschreibt einen Ansatz der Software-Wartung, die mit Reengineering verglichen werden kann (siehe Abb. 6.7). Die Bottom-Up-Methode entspricht dem Reverse-Engineering, wobei die Programmspezifikation des alten Systems neu erstellt wird. Änderungen werden auf dieser mittleren Ebene (vgl. Abb. 6.1) anstatt auf der Programmierungsebene

durchgeführt. Die Top-Down-Methode (vgl. Forward-Engneering) führt anschließend zu einer neuen veränderten Version des Systems.

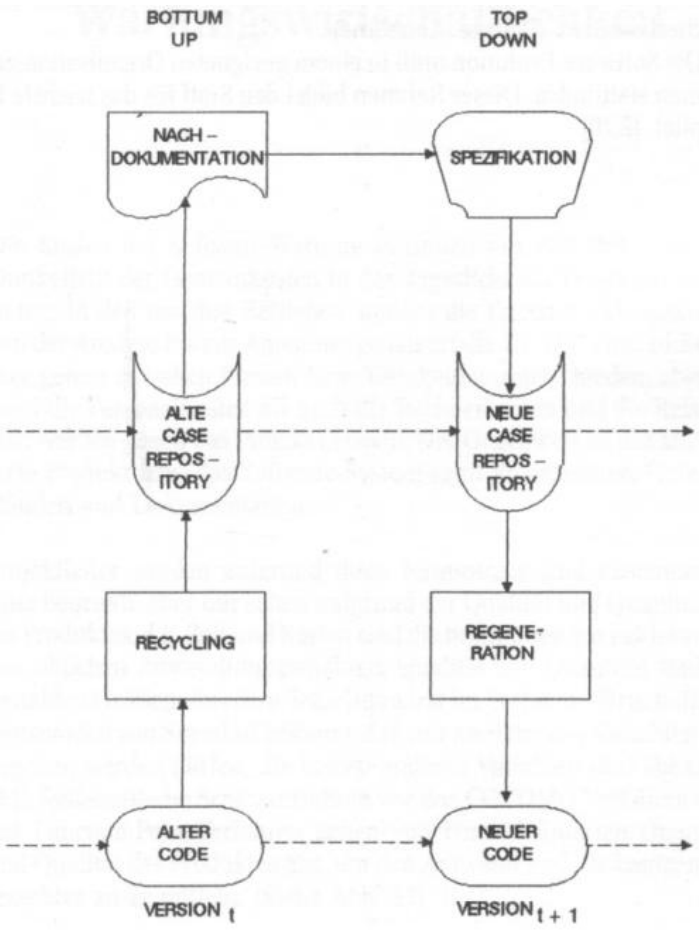


Abbildung 6.7 [Sne91: 81]

6.4.3 Restrukturierungsprozess von Kaufmann

Auch Kaufmann spricht von Reengineering-Prozess [Kau94: 34], wobei er drei Teilgebiete unterscheidet. Zuerst wird in 6.4.3.1 die Bedeutung der Redokumentation von Programmen verdeutlicht. Dann folgt in 6.4.3.2 die Restrukturierung von Programmen und Daten. In 6.4.3.3 wird die Migration von Programmen beschrieben. Zum Schluss steht in 6.4.3.4 eine Stellungnahme zu dem Prozess von Kaufmann.

6.4.3.1 Redokumentation von Programmen

Kaufmann schreibt [Kau94: 34], dass eine wesentliche Voraussetzung für eine effiziente Wartungsdurchführung eine vollständige und konsistente Programmdokumentation ist. Diese soll speziell für die Aufgabe geeigneter Form erstellet werden. Kaufmann unterscheidet zwischen folgenden nicht unbedingt disjunkten Möglichkeiten zur Verbesserung der Programmdokumentation:

- Transformation in eine andere Darstellungsform: Hierzu zählen nachträgliches Formatieren, tabellarische Darstellung der Modulaufrufstruktur, die halbgraphische und die graphische Darstellung (vgl. [Kau94: 34]).

- Generierung nicht vorhandener Dokumentationen: Nicht vorhandene, nicht aktuelle oder nicht konsistente Dokumente sollen durch Neue ersetzt werden (vgl. [Kau94: 35]).

- Extrahieren und Aufbereiten von Teilsichten (program slicing): Diese Art der Nachdokumentation bildet ein mächtiges Instrument zur Steigerung der Wartungseffizienz. Aus der meist großen Anzahl von Einzelobjekten und Strukturen, aus denen sich ein Programmsystem zusammensetzt, werden die für eine spezielle Wartungsaufgabe benötigten Informationen herausgesucht und in einer übersichtlichen Art und Weise dargestellt. Hierzu zählt der Aufbau von Datenreferenztabellen, von Flussdiagrammen für spezielle Daten und von Programmaufrufstrukturen (vgl. [Kau94: 35]).

6.4.3.2 Restrukturierung von Programmen und Daten

Restrukturierung dient zur Standardisierung von Programmelementen (Anweisungen und Daten) und –strukturen (logische Programmstruktur und Datenstruktur) zur Unterstützung des Wartungspersonals, nicht zur Optimierung von technischen Leistungsmerkmalen. Bei der Restrukturierung von Nicht-Kontrollanweisungen steht die Ersetzung von fehlerträchtigen oder unübersichtlichen Konstrukten im Vordergrund. Restrukturierung der Programmlogik dient zur Umordnung der Kontrollstrukturen sowie zur Standardisierung der Verwendung und Reduzierung der Anzahl der Kontrollelemente. Strukturierte Programmierung führt auf der

Ebene von Programmeinheiten (Prozeduren, Blöcke, Unterprogramme) zu hierarchisch geordneten Modulebenen (abstrakten Maschinen), wobei ein Modul einer Ebene nur solche derselben Ebene oder tiefer angeordneter Ebenen aufrufen darf. Auf der Ebene der intramodularen Programmstruktur weisen dadurch die Kontrollanweisungen die Form eines hierarchisch geordneten Baums auf. Die Umwandlung von unstrukturierten in strukturierte Kontrollstrukturen wird häufig mittelbar durch die Umwandlung in eine Flussgraphendarstellung der betreffenden Programme durchgeführt (vgl. Abb. 6.8)

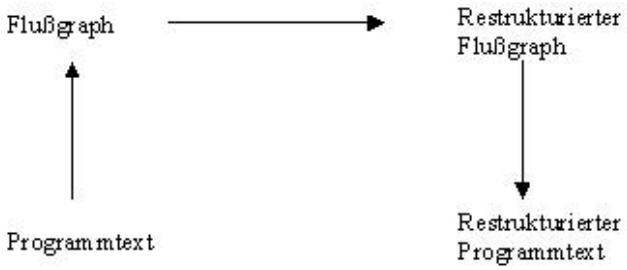


Abbildung 6.8 [Kau94: 37]

6.4.3.3 Migration von Programmen

Bei der Migration erfolgt die Übertragung von Programmen in eine andere Systemumgebung meist auf einer neuen Technologiestufe. Demgemäß müssen bestimmte Komponenten und Teilstrukturen modifiziert oder ersetzt werden. Folgende Migrationen sind zu unterscheiden:

- Datengestaltungssystem
- Benutzerschnittstelle
- Programmiersprachen
- Betriebssystem
- Rechnerarchitektur.

6.4.3.4 Stellungnahme

Kaufmann nennt den oben beschriebenen Prozess Reengineering. Es soll aber betont werden, dass dabei dennoch von einem mittleren Abstraktionsniveau und somit lediglich von einem unvollständigen Reengineering-Prozess gesprochen werden kann. Bei der Dokumentensammlung wird die Programmierebene durch tabellarische, halbgraphische bzw. graphische Darstellung (vgl. 6.4.3.1) verlassen und die Umstrukturierung erfolgt mit Hilfe

von Flussgraphen (vgl. 6.4.3.2), aber es wird das IT-Konzept nicht erstellt. Der Prozess von Kaufmann entspricht einem Reengineering, bei dem die Programmspezifikationsebene nicht erreicht wird.

6.4.4 Zusammenfassung

Hier sollen die Erkenntnisse der Reengineering-Prozesse zusammengefasst werden, die einer Wartung auf dem mittleren Abstraktionsniveau entsprechen.

Ein Ansatz aller Reengineering-Prozesse besteht darin, alle Software-Wartungsaktivitäten auf Programmspezifikationsebene statt auf Code-Ebene durchzuführen. Reverse-Engineering ist das Mittel, die Software-Wartungsaktivitäten auf eine höhere Abstraktionsebene zu verlagern [Mül97: 102]. Die Motivation für einen so komplexen und schwierigen Konversionsprozess ist offensichtlich: Durch den Weg über eine abstrakte, d.h. programmiersprachenunabhängige Repräsentation werden Unzulänglichkeiten des Ausgangsprogramms eliminiert. Das Ergebnis sind effizientere, besser dokumentierte Zielprogramme, die nicht nur die weitere Wartung vereinfachen, sondern auch die Wartungskosten senken.

Gemeinsamkeiten bei allen Reengineering-Prozessen, sowie auch bei den oben genannten Beispielen von Sneed und Kaufmann (vgl. 6.3.2 bzw. 6.4.3), sind die am Anfang bevorstehende Dokumentensammlung, was zum Verständnis der Systeme beitragen soll. Denn nur nach einer globalen Analyse kann das System für eine Restrukturierung vorbereitet werden. Grundlegend ist die Idee, dass eine Transformation nicht mehr auf der Code-Ebene durchgeführt, sondern auf eine höhere Ebene verlagert wird. Hier soll noch mal betont werden, dass in den oben vorgestellten Umstrukturierungsprozess von Sneed und Kaufmann (vgl. 6.3.2 bzw. 6.4.3) dies nicht erfüllt wurde. Sie bleiben nur auf den unteren Ebenen einer Reengineering-Pyramide (vgl. Abb. 6.2) und deswegen handelt es sich dabei nicht um vollständiges Reengeering. Ein vollständiges Reengeering beinhaltet Reverse- und Forward-Engineering. Spezifikationselemente und –strukturen müssen identifiziert, extrahiert und anschließend auf eine höhere Ebene nach neuen Darstellungs- und Strukturierungsregeln übertragen werden. Zum Schluss besteht die Aufgabe, das alte System durch das neue abzulösen. Die Systemmigration ist ein Evolutionsschritt in dem Lebenszyklus der Software, der bis zu einer endgültigen Ablösung in der Regel mehrmals wiederholt werden kann.

Zu einem Reengineering-Prozess gehört nicht die Erstellung und Änderung eines konzeptionellen Modells. Es geschieht auf technischer Ebene. Ein Ansatz zur künftigen effizienten Software-Wartung besteht darin, alle Software-Wartungsaktivitäten auf betriebswirtschaftlicher/konzeptioneller Ebene statt auf IT-Ebene durchzuführen. Software-Reengineering verläuft auf den unteren Ebenen einer Software-Wartungspyramide (vgl. Abb. 6.1). Nur ein vollständiger Wartungsprozess erfasst die konzeptionellen Aspekte, die in 2.1.3, 2.1.4, 3.2.2.2 und 5.2.2 besprochen wurden.

6.5 Wartungsansätze mit höheren Abstraktionsniveau

In 6.1 wurde die Wartungspyramide vorgestellt. Wartungstätigkeiten, die auf der konzeptionellen Ebene ablaufen (vgl. Abb. 6.1), werden hier besprochen. Ein Ansatz um konzeptionelle Aspekte zu erfassen (vgl. 2.1.3, 2.1.4, 3.2.2.2 und 5.2.2) ist, die Wartungstätigkeiten mit Hilfe der Modellierung zu unterstützen. In 6.5.1 soll ein Überblick der Modelle für die Software-Wartung geboten werden. Danach ist in 6.5.2 ein Beispiel für ein Vorgehensmodell der Software-Wartung nach Curth und Giebel zu finden. In 6.5.3 ist das Thema Evolution während der Software-Entwicklung und –Wartung. Ein umfassendes Evolutions-Modell für den gesamten Lebenszyklus eines Systems wird in 6.5.3.1 vorgestellt. Danach werden die Zeiträume auch einzeln besprochen: In 6.5.3.2 wird die Evolution während der Software-Entwicklung diskutiert und in 6.5.3.3 folgt die Software-Evolution, welche während des Zeitraums der Wartung stattfindet. Dazu wird zuerst in 6.5.3.3.1 der Begriff „Software-Evolution“ erläutert. Danach wird auf die Idee von Lehman zurückgegriffen. Die S-, P- und E-Programmtypen bilden die Basis für die Evolutionslehre im Bereich der Software, welche in 6.5.3.3.2 zu finden sind. Danach wird in 6.5.3.3.3 der Software-Evolutionsansatz von Lehman weiter ausgeführt.

6.5.1 Modelle für Software-Wartung

Ein Ansatz in der Software-Wartung ist es, die Tätigkeiten mit Hilfe einer Phaseneinteilung zu unterstützen. In der Software-Entwicklung werden die Tätigkeiten schon lange mit Hilfe von Phasenmodellen unterstützt. Die Software-Entwicklungsmodelle vorzustellen, würde den Rahmen dieser Diplomarbeit sprengen. Weiterführende Literatur findet man in Fachbüchern aus dem Bereich Software-Engineering. Dort sind die gängigen Phasenmodelle, wie das Wasserfallmodell, Spiralmodell und andere vorgestellt, die eine Orientierung für

Wartungsmodelle bieten. Über die Situation der Wartungsmodelle berichtet Sneed [Sne91: 43], dass im Gegensatz zum Software-Entwicklungsprozess, für den ein Überangebot an Vorgehensmodellen zu finden ist, es kaum Modelle für den Software-Wartungsprozess gibt. Er meint, es sei auch lapidar zu behaupten, die Wartung könne genau so ablaufen wie die Entwicklung, da ja immerhin in der Wartung die gleichen Phasen wie in der Entwicklung durchlaufen würden. Deswegen eignen sich die Phasenmodelle der Entwicklung nicht vollständig für die Wartung. Sneed veranschaulicht die Problematik durch ein Beispiel: Es erfordert unterschiedliche Vorgehensweisen, eine Straße zu bauen oder eine vorhandene Straße zu reparieren oder zu erweitern. Im zweiten Fall kommt es vor allem darauf an, so viel wie möglich von der alten Bausubstanz zu bewahren und so wenig wie möglich durch die Wartungsmaßnahmen zu zerstören, bei gleichzeitiger Aufrechterhaltung des laufenden Betriebes. Es werden andere Methoden und zum Teil andere Werkzeuge eingesetzt. Vor allem wird jedoch ein neues Phasenmodell vorausgesetzt.

6.5.2 Das Maintenance-Engineering-Modell nach Curth und Giebel

„Software-Engineering“ beschreibt das ingenieurmäßige Vorgehen während der Software-Entwicklung. Parallel dazu führen Curth und Giebel den Begriff „Maintenance Engineering“ ein [Cur89: 114]. In ihrem Buch „Management der Software-Wartung“ zeigen sie, wie die Wartung organisatorisch gestaltet werden sollte. Die Wartungstätigkeiten sollen durch den Rahmen des Managements geregelt werden. Diesen Rahmen bilden das Maintenance-Management, die Qualitäts-Sicherung und die Dokumentation (siehe Abb. 6.9).

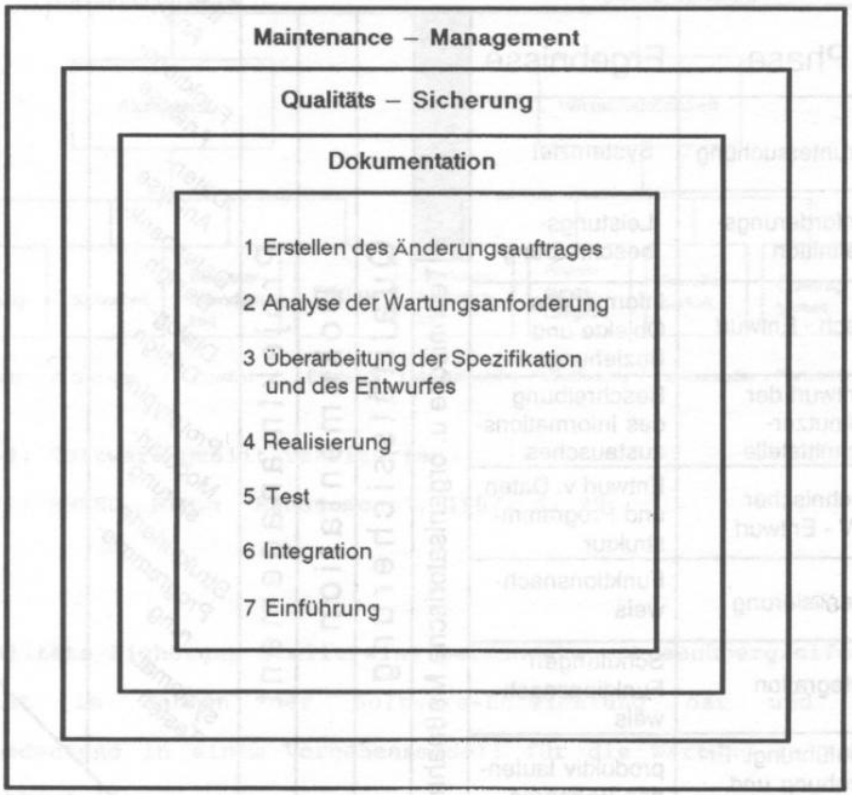


Abbildung 6.9 [Cur89: 114]

In „Management der Software-Wartung“ von Curth und Giebel stehen die organisatorischen Probleme der Wartung im Mittelpunkt. Sie versuchen, diese durch ein geeignetes Management zu lösen. Das „Maintenance-Engineering-Modell“ beschreibt einerseits den organisatorischen Ablauf der Wartungstätigkeiten, andererseits die einzelnen Phasen der Wartung. Die Phasen bilden den Kern dieses Modells, der sich stark an den Software-Entwicklungsphasen orientiert. Die Reihenfolge: Analyse, Entwurf, Realisierung, Test und Einführung ist ein typischer Ablauf der Software-Entwicklung. Eine andere Parallelität besteht mit dem Reengineering: Die Überarbeitung der Spezifikation und des Entwurfes ist ein Prozess, der bei Reengineering stattfindet (vgl. Abb. 6.2). In 6.2.2. wurden dazu bereits die wichtigsten Schritte zusammengefasst. Bei einem Reengineering-Prozess steht zum Schluss die Aufgabe, das alte System durch das neue abzulösen. In der obigen Abbildung wird dementsprechend die Integration aufgeführt. Das „Maintenance-Engineering-Modell“ ist aber mehr als nur Reengineering. Es ist nicht auf die technische Ebene reduziert. Das „Erstellen des Änderungsauftrages“ und die „Analyse der Wartungsanforderung“ beinhalten die Einbeziehung der konzeptionellen Aspekte. Das Modell entspricht somit einer vollständigen Wartungspyramide, wie in Abbildung 6.1 zu sehen ist.

6.5.3 Evolution während der Software-Entwicklung und –Wartung

Wie der Titel andeutet, kann eine Evolution von Software während des Entwicklungszeitraums und während der Wartung erfolgen. In dieser Arbeit soll **vor dem Einsatz** eines Systems von „der evolutionären Software-Entwicklung“ und **ab dem Einsatz** von „Software-Evolution“ die Rede sein. Ein umfassendes Evolutions-Modell für den gesamten Lebenszyklus eines Systems wird in 6.5.3.1 vorgestellt. Danach werden die Zeiträume auch einzeln besprochen: In 6.5.3.2 wird die Evolution während der Software-Entwicklung diskutiert. In 6.5.3.3 folgt die Software-Evolution, welche während des Zeitraums der Wartung stattfindet. Dazu soll zuerst in 6.5.3.3.1 der Begriff „Software-Evolution“ erläutert werden. Danach wird auf die Idee von Lehman zurückgegriffen. Die S-, P- und E-Programmtypen bilden die Basis für die Evolutionslehre im Bereich der Software. In 6.5.3.3.2 werden diese drei grundlegenden Programmtypen vorgestellt. Danach wird in 6.5.3.3.3 der Software-Evolutionsansatz von Lehman weiter ausgeführt.

6.5.3.1 Evolutionäre Software-Entwicklung und –Wartung nach Tom Gilb

Tom Gilb beschreibt (zitiert nach [Sne91: 43]) ein übergreifendes Modell für die Entwicklung und Wartung von Software (siehe Abb. 6.10). Das Modell „Evolutionäre Software-Entwicklung“ von Gilb umfasst die Entwicklung und die gesamte Lebensdauer der Software. D.h. Tom Gilb unterscheidet nicht zwischen der Evolution vor bzw. ab dem Einsatz von Software. Im Gegensatz zu Gilb werden in dieser Arbeit die zwei Zeiträume voneinander unterschieden, wie es in 6.5.3 beschrieben wurde. Demnach ist Abbildung 6.10 sowohl das Modell für die evolutionäre Software-Entwicklung wie auch für die Software-Evolution, die demnächst beide ausführlich besprochen werden sollen.

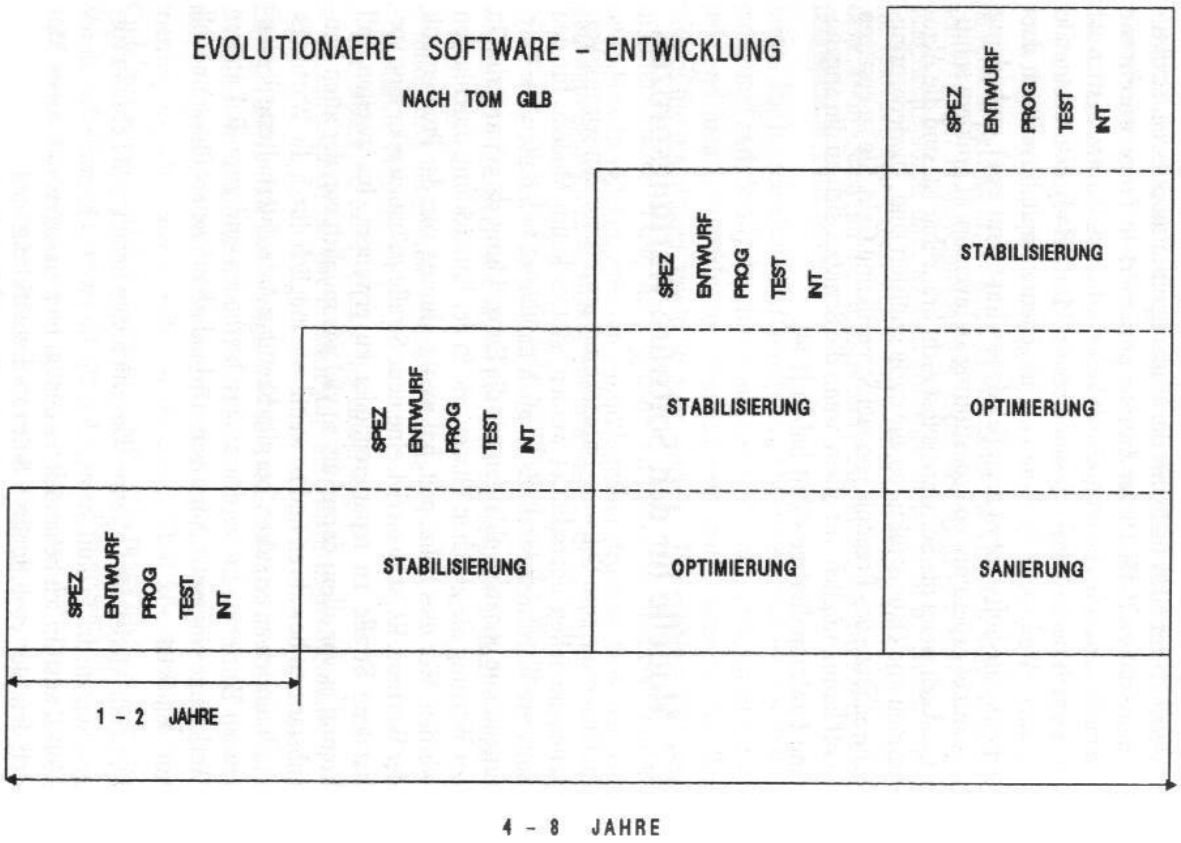


Abbildung 6.10 [Sne91: 44]

6.5.3.2 Evolutionäre Software-Entwicklung

Es wurde bereits in 6.5.3 betont, dass eine Evolution von Software bereits in dem Entwicklungszeitraum stattfinden kann. Es gibt viele Software-Entwicklungsmethoden, die Wartung erleichtern können. Abbildung 6.11 veranschaulicht, wie die Entwicklung die Grundlagen zu einer erfolgreichen Wartung sichert:

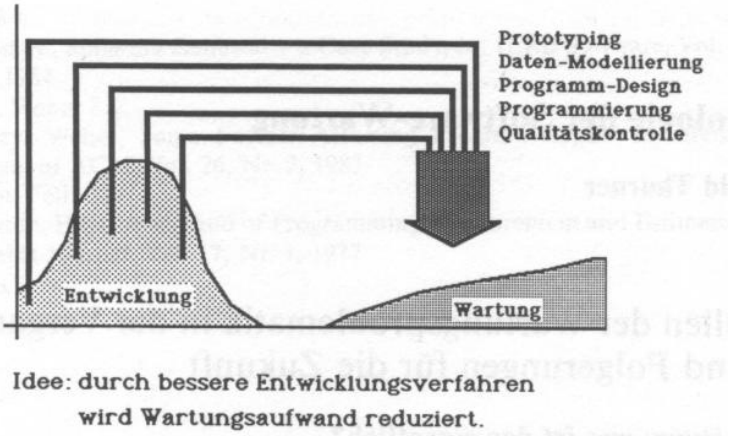


Abbildung 6.11 [Thu88: 146]

Thurner [Thu88: 145] beschreibt welche Zielsetzungen bis jetzt in den meisten Software-Entwicklungen angestrebt wurden, die jedoch unrealistisch sind und daher nicht zu dem erwünschten Ergebnis führten:

> „Nicht nur die Erfassung der Anforderungen, auch die Ansätze zur Realisierung sind darauf ausgerichtet, spätere Änderungen auszuschließen: Das Datenmodell soll die abschließende umfassende Beschreibung der Daten sein. Der optimale Programm-Entwurf soll die abschließend beste Programmstruktur festlegen. Das gesamte Gebäude basiert auf der unerschütterlichen Annahme einer vollständigen Lösbarkeit des Softwareproblems in einem Anlauf" [Thu88: 145].

Der obige Ansatz ist zum Scheitern verurteilt, weil eine vollständige Anforderungsspezifikation immer nur der Ist-Situation entspricht. Eine kontinuierliche Weiterentwicklung des IT-Systems ist nicht zu vermeiden. Im Laufe des Software-Lifecycles verändert sich die Umwelt und eine Anpassung wird erforderlich. „Spätere Änderungen" können nie vollständig ausgeschlossen und sollten schon vorweg eingeplant werden. Wenn sich die Umwelt ändert, muss das IT-System angepasst werden. Die Vielzahl von Veränderungen konzeptioneller Herkunft wurde bereits in 3.2.2.2 dargestellt. Diese tragen dazu bei, dass eine immer größer werdende Anzahl von ursprünglichen Annahmen, die bei der Entwicklung getroffen wurden, ungültig wird.

Eine „vollständige Lösbarkeit … in einem Anlauf" bleibt nur eine „Annahme". Um in der Entwicklung bessere Ergebnisse zu erzielen, empfehlt Thurner [Thu88: 146] die Maßnahme Prototyping. Prototyping ist eine Methode, auf die sich die evolutionäre Software-Entwicklung stützt. Demnach erfolgt eine evolutionäre Software-Entwicklung stufenweise (siehe Abb. 6.12). Zuerst wird ein Prototyp erstellt. Dieser wird dann kontinuierlich weiterentwickelt. Die Spezifikation des IT-Systems wird immer mit den Vorgänger-Versionen abgeglichen und entsprechend einem neuen Konzept erweitert. Anhand dessen kann dann die neue Produktionsversion aufgebaut werden (vgl. [Sne91: 74]).

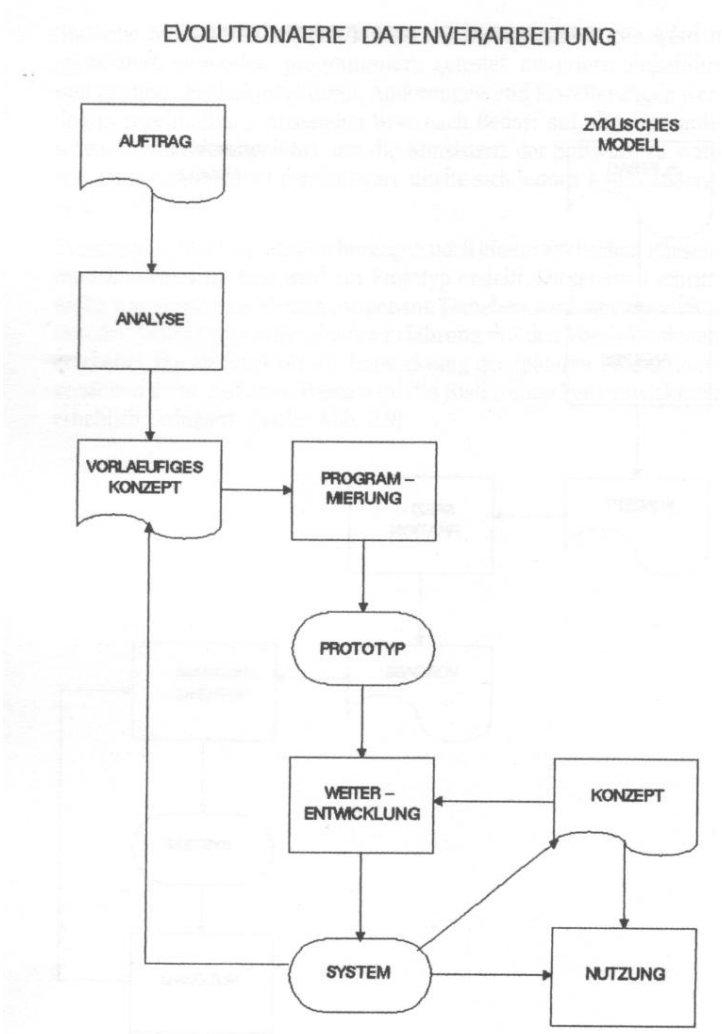


Abbildung 6.12 [Sne91: 76]

In der Abbildung 6.12 ist zu erkennen, dass die Anforderungen an ein IT-System durch Prototyping während der evolutionären Software-Entwicklung erforscht werden. Somit kann die aktuelle Situation möglichst vollständig erfasst werden. Die zukünftige Situation wird immer anders sein als die gegenwärtige. Diese wird später während der Software-Evolution miteinbezogen.

Zusammenfassend gilt für die Software-Entwicklung, dass durch das evolutionäre Vorgehen während der Entwicklung dafür Sorge zu tragen ist, dass die Anforderungen der aktuellen Situation so vollständig wie möglich erfasst werden. Bei den Anforderungen sollen auch die zukünftigen Entwicklungen, die vorhersehbar sind, mitberücksichtigt werden. Die Entwicklung kann spätere Änderungen nicht außer Acht lassen. Ein geeigneter Entwurf bzw. eine geeignete Architektur soll sicherstellen, dass Anpassungen und Erweiterungen leicht

durchführbar werden. Somit sind in der Zukunft entstehende Anforderungen bei der späteren Weiterentwicklung leichter realisierbar.

6.5.3.3 Software-Evolution

Software-Evolution wird in dieser Arbeit für die Evolution während der Software-Wartung verstanden. Um einen tieferen Einblick zu ermöglichen, soll in 6.5.3.3.1 zuerst der Begriff „Software-Evolution“ erläutert werden. M. M. Lehman wird von vielen der „Vater der Software-Evolution“ genannt. In 6.5.3.3.2 wird auf seine Idee zurückgegriffen: Es werden die S-, P- und E-Programmtypen vorgestellt. Danach soll in 6.5.3.3.3 der Software-Evolutionsansatz von Lehman weiter besprochen werden.

6.5.3.3.1 Definition

Noch gibt es keine genormte Definition für „Software-Evolution“. Somit ist der Begriff nach meinen Recherchen in keinem Computer-Lexikon zu finden. Nach einer allgemeinen Definition ist Evolution eine „langsame, kontinuierlich fortschreitende Entwicklung“ [Bro97].

Obwohl es keine genormte Definition für die „Software-Evolution“ gibt, wird der Begriff bereits etwa ab 1970 benutzt ([Leh69], [Leh72: 503] zitiert nach [web6.1]). Seitdem haben sich viele Wissenschaftler bemüht, eine Begriffserklärung aufzustellen. Besonders im Internet konnte ich viel zu diesem Thema finden.

Nach dem Skript von Harald Gall an der Universität Zürich wird unter Software-Evolution folgendes gelehrt:

> „Software Evolution bezeichnet den Prozess der Veränderung eines Softwaresystems von der Erstellung bis zur Stilllegung. Umfasst: Entwicklung, Wartung, Migration, Stilllegung“ [web6.2].

An der Universität Wien wurde in einem Vortrag von 2002 die folgende Definition vorgestellt:

> „Software Evolution is the way a software system reacts to changing requirements. Software evolution occurs when software artefacts change.

> Software evolution refers to the sequence of changes that software goes through from its first release until its retirement.
> Software evolution is the systematic process of extending and adapting systems, without starting from scratch.
> The realized history of the software system during its lifetime" [web6.3].

In einer Quelle von David Hearnden wird Rajlich zitiert:

> "Software evolution is the changes in software requirements that result in consequent changes of the software" [Raj97 zitiert nach [web6.4]].

Eine zweite weiterführende Definition von David Hearnden ohne Quellenangabe ist:

> "Software evolution is the process of making incremental change to software artefacts, while preserving the relationships between artefacts" [web6.4].

Das "Research Institute in SW Evolution" definiert Software-Evolution als:

> "The set of activities, both technical and managerial, that ensures that software continues to meet organisational and business objectives in a cost effective way" [web6.5].

Ned Chapin (1999) definiert Software-Evolution als:

> "The application of Software Maintenance activities and processes that generate a new operational software version with a changed customer-experienced functionality or properties from a prior operational version (…) together with the associated quality assurance activities and processes, and with the management of the activities and processes" [web6.5].

Manny Lehman und Juan Ramil (2000) definieren Software-Evolution als:

> "All programming activity that is intended to generate a new software version from an earlier operational version" [web6.5].

6.5.3.3.2 Programmtypen

Die „Software-Evolution“ von Lehman basiert auf Beobachtungen der von ihm definierten Programmtypen. Er unterscheidet drei Klassen von Programm-Systemen: S-Systeme, P-Systeme und E-Systeme (zitiert nach [Sne91: 66]).

- S-Systeme (S ist eine Abkürzung für engl. „specifiable“) sind statisch; es sind so genannte Wegwerfprogramme. Es handelt sich dabei um einfache Programmsysteme, deren Funktion sich aus einer formalen Spezifikation ableiten lässt. Qualitativ lassen sie sich danach beurteilen, wie weit sie ihre Spezifikation korrekt wiedergeben. Wenn die Programme einen Ausschnitt aus der realen Welt korrekt abbilden, kommt es darauf an, ob dieser Veränderungen ausgesetzt ist oder konstant bleibt. Verändert sich die reale Welt, ist die Spezifikation veraltet und damit auch das Programm. Wenn die Spezifikation der S-Systeme sich ändert, werden sie durch neue Programme ersetzt.

- P-Systeme (P ist eine Abkürzung für engl. „problem solution“) sind nur Approximationen der realen Welt. Sie sind komplexe Systeme und daher ist eine korrekte Spezifikation nie möglich, weil sie eine inkonsistente und unstabile Realität abbilden. Somit sind P-Systeme änderbar und im Laufe ihres Lebens werden sie in mehreren Versionen erstellt. Eine ständige Anpassung wird durch einen Rückkopplungsmechanismus zu der laufenden Anwendung gefördert. Dennoch lassen sich Abweichungen zwischen den Programmen, der Spezifikation und den Anforderungen nicht vermeiden.

- E-Systeme (E ist eine Abkürzung für engl. „embedded“ = eingebettet) sind hypothetische Programmsysteme. Sie werden anhand Prognosen über die zukünftige Realität erstellt. Es handelt sich dabei nicht um Abbildungen sondern um eine Projektion, die modelliert und spezifiziert wird. Das Prognosemodell erfordert unvermeidlich Korrekturen. Die Spezifikationen und Programme werden immer wieder geändert. Ein Rückmeldewesen ist hier unerlässlich. Die gesammelten Erfahrungen und die geänderten Anforderungen tragen zur neuen Prognoseerstellung bei. Im Lebenszyklus der E-Systeme entstehen zahlreiche Versionen, welche die Wandlung der realen Welt abbilden. Die ersten Versionen sind nur sehr ungenaue Systeme, dienen aber als Grundlage für die späteren. Jede neue Version trägt dazu bei,

die Bedürfnisse der Anwender zu entdecken, um auf diese reagieren zu können. Wichtig ist, dass die Programme flexibel und ausbaufähig sind, damit sie in jede beliebige Richtung ausgebaut werden können. So entsteht eine evolutionäre Software-Entwicklung.

Die Unterscheidung zwischen S-, P- und E-Programmen verdeutlicht, dass manche Programme einer Evolution unterliegen. Im Gegensatz zu den statischen S-Programmen, unterliegen P- und besonders die E-Programme einer ständigen Veränderung. Eine wesentliche Ursache für die Evolution von P- und E-Programmen ist, neben der Veränderung der Umwelt, die Evolution des Wissens. Die reale Welt ist nicht statisch und auch unser Wissen über sie wird ständig erweitert. Dabei kann nicht ausgeschlossen werden, insbesondere bei E-Programmen, dass das Wachstum des Wissens über einen bestimmten Diskursbereich gerade durch die Nutzung einer einschlägigen Software ausgelöst wird. Deshalb kann man sagen, dass E-Programme selbst zur Evolution beitragen (Rückkopplung). Das Evolutionsmodell von Lehman ist für P- und besonders für E-Systeme geeignet. Die früheren Versionen tragen zum Wissenserwerb bei. Durch eine ständige Rückkopplung des Wissens werden die weiteren Versionen erarbeitet. Die kontinuierliche Weiterentwicklung dieser Systeme ist die eigentliche Evolution der Software.

6.5.3.3.3 Der Software-Evolutionsansatz von Lehman

Lehman schlägt vor, den Begriff „Wartung“ durch den Begriff „Evolution“ zu ersetzen (zitiert nach [Sne91: 45]), da seiner Meinung nach komplexe Software-Systeme sowieso nie fertig sind. Die kontinuierliche Weiterentwicklung der Software soll dem Alterungsprozess entgegenwirken. Das Modell von Gilb dient an dieser Stelle auch als Modell für die Software-Evolution (siehe Abb.6.10): Nach dem Einsatz der ersten Version des Systems wird dieses nach Bedarf weiterentwickelt. Als Grundlage dient dabei jeweils die zuletzt eingesetzte Systemversion, deren Spezifikation erweitert wird. Anhand dieser Spezifikation kann dann die neue Produktversion aufgebaut werden (vgl. [Sne91: 74]), die solange im Einsatz bleibt, bis eine neue Version erstellt oder das System vollständig saniert wird.

Ein weiteres Modell soll die Idee von Lehman veranschaulichen. Das Iterative-Enhancement-Modell von Basili (siehe Abbildung 6.13) entspricht dem evolutionären Vorgehen während der Wartung [Bas90 zitiert nach [Lehner91: 14]].

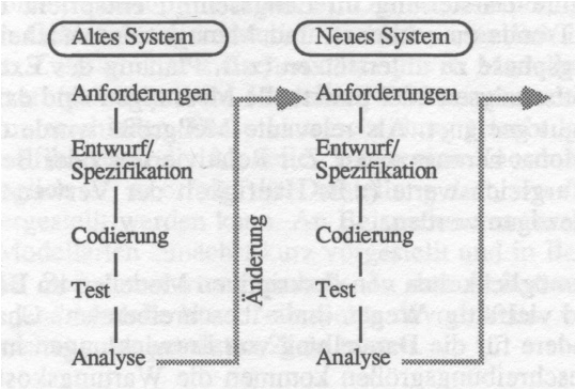


Abbildung 6.13 [Bas90 zitiert nach [Lehner91: 14]]

Im Gegensatz zu dem Quick-Fix-Modell (vgl. Kapitel 4.2.1 und Abbildung 4.2) werden hier Änderungsmaßnahmen nicht auf der technischen, sondern auf der konzeptionellen Ebene durchgeführt (vgl. Abb. 6.1).

Balzer [Bal79 zitiert nach [Sne91: 45]] unterstützt die Meinung von Lehman in seinem Aufsatz über das „Interwining von Spezifikation und Implementation“. Er meint, es sei falsch, zu glauben, Anforderungen könnten jemals vollständig definiert werden. Denn mit jedem Schritt der Verfeinerung werden sie umdefiniert. Ein Fachkonzept reflektiert demnach immer nur einen gewissen Stand der Dinge.

> „The major problem in requirements specification may not be the development of a complete, consistent and inambiguous specification, prior to design, but, rather, the evolution of requirements which allow a timely response of the software to organizational change" [Bal81 zitiert nach [Sne91: 45]].

Auch Curth und Giebel schreiben [Cur89: 101], dass es nicht das Ziel sein möge, eine 100%-Lösung in die Spezifikation einzubringen, sondern es viel mehr anzustreben sei, ausgehend vom Systemdesign ein klares Modulkonzept zu erstellen, das es erlaubt, Module problemlos zu ändern, zu erweitern und neu einzufügen.

Gilb ergänzt Idee von Lehman [Gil88 zitiert nach [Sne91: 80]]: Er meint das bisherige Projektdenken soll abgelöst werden: Es gilt nicht, ein Projekt als eine einmalige Anstrengung zu betrachten, damit ein neuer Ist-Zustand erreicht werden kann. Ein endgültiger Zustand ist weder erreichbar noch wünschenswert. Nach Gilb, Boehm und anderen (vgl. [Sne91: 79]) sind Projekte nur Zwischenschritte, um das bestehende System in einen besseren Zustand zu

versetzen. Dabei gibt es nur Zwischenzustände auf dem Wege zu einem unbekannten Bestimmungsort. Somit wird ein komplexes Software-System Schritt für Schritt über viele Jahre hinweg aufgebaut. Das bestehende System dient als Grundlage, d.h. es wird von dem Bestehenden ausgegangen.

7 Beispiele

Dieses Kapitel listet Beispiele auf, welche die bisherigen Erkenntnisse vertiefen sollen. Es werden einzelne Beispiele genannt, die zeigen, dass Fehler in der Entwicklung zu schwerwiegenden Problemen der Wartung führen. In 7.1 wird das Beispiel zum Jahrtausendwechsel vorgestellt. Dazu wird zuerst in 7.1.1 erklärt, was das eigentliche Problem des Jahrtausendwechsels war und danach soll in 7.1.2 die Frage beantwortet werden, ob die Probleme vermeidbar gewesen wären. In 7.2 folgt ein mit dem Jahrtausendwechsel vergleichbares Problem, ein Beispiel des Betriebsystems UNIX. Ein weiteres Beispiel, nämlich die Euro-Umstellung, ist in 7.3 aufgeführt. Ein ähnliches Problem wie die Euro-Umstellung, ist in 7.4 zu finden. Es handelt sich um die Umstellung des Postleitzahlsystems in Deutschland. Dazu wird in 7.4.1 zuerst die Ausgangsituation beschrieben. Danach folgt in 7.4.2 eine Darstellung der Umstellungsprobleme. Zum Schluss ist in 7.4.3 eine Stellungnahme zu diesem Beispiel zu lesen. In diesen gesamten Beispielen hätten die Probleme geringer ausfallen können, wenn bei der Software-Entwicklung die zukünftige Entwicklung des Umfeldes mit eingeplant gewesen wäre. In 7.5 soll aufgeführt werden, woran bei einem konzeptionellen Entwurf von Software-Systemen zu denken ist. Dazu sind in 7.5.1 die wichtigsten Gedanken zusammengefasst. Ein Beispiel in 7.5.2 soll zeigen, wie die Ideen in der Praxis zu realisieren sind. Die daraus gewonnenen Erkenntnisse werden in 7.5.3 zusammengefasst.

7.1 Beispiel für vorhersehbare zukünftige Entwicklungen: Das „Jahr-2000-Problem“

Das Beispiel des Jahr-2000-Problems soll verdeutlichen, mit welchen Auswirkungen man rechnen muss, wenn bei der Software-Entwicklung die zukünftigen Entwicklungen nicht berücksichtigt werden. Das Ziel, Anforderungen vollständig zu erfassen, kann nicht erfüllt werden, wenn die vorhersehbare Entwicklung der Zukunft außer Acht gelassen wird. Hierzu soll zuerst in 7.1.2 erklärt werden, was unter dem Jahrtausendproblem zu verstehen ist. Danach wird in 7.1.2 beschrieben, wie man die Probleme bereits im Vorfeld erkennen und damit vermeiden hätte können.

7.1.1 Was ist das Jahr-2000-Problem?

In vielen Informatiksystemen werden Jahreszahlen in der auch im Alltag sehr gebräuchlichen, zweistelligen Kurzform – also beispielsweise “98” statt “1998” – repräsentiert, verarbeitet und gespeichert. Solche zweistelligen Darstellungen können, wie beim Übergang vom Jahr 1999 auf das Jahr 2000, zu Problemen führen. So liefert beispielsweise die Berechnung des Alters einer im Jahre 1959 geborenen Person im Jahre 1997 (noch) ein korrektes Ergebnis (“97” – “59” = “38”). Dieselbe Berechnung führt aber im Jahr 2000 zu einem falschen Resultat (“00” – “59” = “-59”). Es treten aber nicht nur Schwierigkeiten bei Zeitraumberechnungen, sondern beispielsweise auch beim Sortieren und Mischen von Daten auf (“1997” < “2000” aber “00” < 97”). Die “Breite” des Jahr-2000-Problems ist sehr weit, worauf hier nicht im Einzelnen eingegangen wird (vgl. [web7.1]).

7.1.2 War das Jahr-2000-Problem vermeidbar?

Um die Frage zu beantworten, sollen hier die Hintergründe des Problems erläutert werden. Es sind eine ganze Reihe von Gründen, die zu dieser Situation geführt haben (vgl. [web7.1]):

- Als eine wesentliche Ursache haben sicher knappe Ressourcen bei der Entwicklung (Speicherplatz, Rechenzeit, ...) eine Rolle gespielt. Dies bildet – in Kombination mit einer oft anzutreffenden Fehleinschätzung der Lebensdauer von Anwendungen – ein an sich einleuchtendes Argument. Das Einsparen von zwei Bytes hat aber schon seit längerem nicht mehr dieselbe Bedeutung und reicht als Rechtfertigung allein sicher nicht aus.

- Ein weiteres gewichtiges Argument ist in der Mehrfachnutzung von Programmteilen und Daten zu sehen. Namentlich mit dem Aufkommen relationaler Datenbanksysteme – die eine recht weitgehende Datenunabhängigkeit von Anwendungen unterstützen – war (und ist) es wirtschaftlich nicht vernünftig (und auch nicht gewünscht!), bestehende Datenbestände und darauf operierende Programme bei Einführung einer neuen Anwendung anzupassen. Dazu kommt, dass eine zweistellige Darstellung zu einer Vereinfachung von Ein-/Ausgabeschnittstellen führt (effizientere Dateneingabe, übersichtlichere Darstellung auf Masken und Listen u.a.m.).

- Lange Zeit fehlende Standards zur Darstellung von Kalenderdaten (so wurde beispielsweise die ISONorm 8601 erst 1988 in Kraft gesetzt) haben zudem zu einer Vielzahl verschiedener Darstellungsarten von Kalenderdaten geführt.

- Weitaus schwerer fällt es allerdings, einleuchtende Gründe für entsprechende Sachverhalte im Bereich der systemnahen Software bzw. bei Hardware / Firmware zu finden. So ist es beim besten Willen nicht einzusehen, warum ein Datentyp "Datum" in einem COBOL-Compiler nur zweistellige Jahreszahlen unterstützt [IBM97 zitiert nach [web7.1]] oder warum sich die Uhr eines Videorekorders nicht auf ein Datum nach dem 31.12.1999 einstellen lässt.

Die Frage, ob das Jahr-2000-Problem vermeidbar gewesen wäre, kann damit beantwortet werden, dass eine Vielzahl fehlerhafter Planungen die Situation verschlimmert hatte. Ein mehr auf die Zukunft ausgerichtetes Denken hätte das Problem sicherlich deutlich verringert. Die anfängliche Speicherplatzknappheit rechtfertigt zwar die ursprünglich zweistellige Darstellung der Jahreszahlen, aber eine spätere Umstellung auf vierstellige Jahreszahlen hätte viel früher stattfinden sollen. Es ist aber leider eine allzu oft typisch menschliche Verhaltenwiese, die Probleme so lange unbeachtet zu lassen, bis diese schließlich nicht mehr zu ignorieren sind. Dann fallen aber die Konsequenzen meistens viel schwerwiegender aus, wie es auch beim Jahr-2000-Problem der Fall war, als wenn man sich schon früher mit dem Problem auseinander gesetzt hätte. Ein zukunftsorientierter Systementwurf ist notwendig, um solchen Problemen aus dem Weg gehen zu können. Weiterhin ist die Software-Evolutionslehre dadurch hilfreich, dass in dem Entwurf eine spätere Änderung mit eingeplant werden kann. In diesem Beispiel hätten, bei einer gleich bleibenden zweistelligen Darstellungsform, wenigstens die Vorbereitungen auf eine spätere Umstellung stattfinden sollen.

7.2 Beispiel für vorhersehbare zukünftige Entwicklungen: Das Betriebssystem UNIX

Der Jahrtausendwechsel war speziell bei den Betriebssystemen Microsoft DOS und Windows ein Problem. Ein weiteres Betriebssystem, das UNIX, wurde von den Entwicklern mit der nötigen Weitsicht geplant [web7.2]. So wird UNIX meist auf Computern mit 32bit Architektur eingesetzt. Aus diesem Grund verwaltet UNIX das Datum in Form einer 32bit Integer-Zahl. Diese Art der Datumsinterpretation nennt man auch serielles Datum. In einer

Integer-Variablen werden die Sekunden seit dem 01.01.1970, der Geburtsstunde von UNIX, fortgeschrieben. Ein Überlauf dieser Variablen findet am 19.01.2038 um 3:14:07 Uhr statt, denn dann ist der maximale Wert, den diese Variable speichern kann, erreicht. Für UNIX gab es also kein Jahr-2000-Wechsel-Problem. Eine ähnliche Situation könnte jedoch auch 2039 auftreten. Allerdings ist es sehr unwahrscheinlich, dass dann noch 32-bit-Systeme in Betrieb sind. Die Umstellung der Betriebssysteme auf 64 Bit und damit auch auf 64-Bit-Rechner ist schon in vollem Gange. Damit würde die dann verbreitete 64 Bit Zählung bis „in alle Ewigkeit“ reichen, denn 2^{64} Sekunden entsprechen ca. 292.27 Billionen Jahren [web7.3].

7.3 Beispiel für unvorhersehbare zukünftige Entwicklungen: Die Euro-Umstellung

Im Gegensatz zu dem Beispiel des Jahr-2000-Problems, welches ein korrektives Wartungsproblem war, handelt es sich bei der Euro-Umstellung um adaptive Wartung (vgl.2.2). D.h. es war bei vielen Entwicklungen tatsächlich nicht eindeutig vorhersehbar, dass die Währung „Euro“ eingeführt wird. Es bleibt dennoch festzustellen, dass besonders bei Einzelentwicklungen für die Umstellung keine notwendigen Vorkehrungen getroffen wurden, obwohl bereits die wirtschaftlichen Hintergründe den Währungswechsel verdeutlicht hatten. Weiterhin bleibt anzumerken, dass ein Systementwurf, der Änderungen leicht zulässt, besonders dann wichtig ist, wenn das Wissen über wirtschaftliche oder sonstige Entwicklungen nicht bekannt ist. Zukunftsprognosen sowie Requirements Engineering sind dann bei der Planungsphase hilfreich. Dieses Beispiel verdeutlicht, dass Änderungen nie ausgeschlossen werden können.

7.4 Beispiel für unvorhersehbare zukünftige Entwicklungen: Das Postleitzahlsystem

Ein weiteres Beispiel, das eher mit der Euro-Umstellung verglichen werden könnte, war etwa die Umstellung des Postleitzahlsystems in Deutschland [web7.1]. D.h.: Es handelt sich um einen Fall, bei dem die Entwicklungen nicht unbedingt vorhersehbar waren, wobei aber deutlich wird, wie wichtig es ist, eine Software für spätere Änderungen flexibel zu halten. Die neuen Postleitzahlen wurden am 1. Juli 1993 eingeführt. Das vorherige System war seit 1961 im Einsatz. Ein neues System war aber nach der deutschen Einheit notwendig, sonst wären ca. 800 Postleitzahlen in West- und Ostdeutschland doppelt vorhanden gewesen [web7.4]. Die Umstellung 1961 hatte nicht die gleiche Auswirkung wie 1993. Diesmal war die IT-Welt mit vielen Problemen konfrontiert. Zuerst wird in 7.4.1 die Ausgangssituation beschrieben.

Danach folgt in 7.4.2 eine Darstellung der Umstellungsprobleme. Zum Schluss ist in 7.4.3 eine Stellungnahme zu diesem Beispiel zu lesen.

7.4.1 Ausgangssituation

Die Umstellung vorhandener Datenbestände und Programme war für viele Unternehmen aus zahlreichen Gründen ein komplizierter Vorgang. Die Menge der Postleitzahlen stieg von 5300 auf rund 34 000 [web7.5]. Die vorher vierstelligen Codes wurden auf fünfstellige Zahlen erweitert. Für die Umstellung stellte zwar die Post Leitdateien zur Verfügung [web7.5], aber auch diese führten zu weiteren Problemen. Laut Kritiken waren die Umstellungsdateien der Bundespost fehlerhaft, so dass eine komplette Eins-zu-eins-Übertragung praktisch ausgeschlossen war [web7.4]. Es gab z.B. Strassen, die in den Dateien der Post gar nicht aufgeführt waren. Besonders die ostdeutschen Strassen waren nicht vollständig, weil hier zahlreiche Namensänderungen durchgeführt wurden. Somit war das Verzeichnis nicht komplett und eine Reihe von Anschriften ließ sich nicht den einzelnen Postleitzahlbezirken zuordnen [web7.4].

Eine schwierige Situation ergab sich mit den Postleitzahlen in Berlin, wo etwa zwei Jahre vor der Postleitzahlumstellung die Menge der Postzustellämter reduziert worden war. Diese Reform hat aber nur auf dem Papier stattgefunden [web7.4]. So gab es für eine Reihe von Postzustellbezirken in der Bundeshauptstadt zwei gültige vierstellige Postleitzahlen: eine amtliche, die postintern offiziell benutzt wurde und eine, mit der Zusteller und Kunden arbeiteten. Die Einführung der fünfstelligen Postleitzahlen war für diesen Bereich kaum möglich, da in den Umstellungsdateien der Post nur die amtlichen Postdaten zur Verfügung gestellt wurden.

7.4.2 Schwierigkeiten bei der Umstellung

Die Änderungen erschöpften sich nicht im Einsatz der Umstellungsprogramme der Post. Auch wenn man von den oben genannten Mängeln der bereitgestellten Dateien absieht, mussten sich die Unternehmen mit vielseitigen Wartungstätigkeiten beschäftigen. Um die Umstellungsdateien einsetzen zu können, mussten die In- und Output-Schnittstellen mit den richtigen Daten versorgt werden. Ein automatisches Ändern setzte voraus, dass die Schreibweise der Altadressen mit der in den Umstellungsdateien übereinstimmte. Veraltete

Bezeichnungen, unterschiedliche Abkürzungen, Schreibfehler etc. mussten die Unternehmen vorher bereinigen [web7.6]. Die einzelnen Unternehmen waren je nach Situation (Menge und Qualität der Adressen, Struktur von Programmen und Daten) mit zahlreichen, oft nicht früh genug erkannten Fußangeln konfrontiert [web7.6]. Dezentrale Adressenbestände zum Beispiel in PCs erforderten spezielle Konvertierungen oder manuelle Umstellung. Betroffen waren in Datenbanken und auch in Standardsoftware gespeicherte Adressen für Briefaktionen [web7.6]. Es waren nicht allein die Adressbestände, sondern auch komplexe Anwendungen betroffen. So mussten etwa die jeweiligen Feldlängen für Orts- und -Straßennamen den geänderten Vorgaben des Postdienstes angepasst werden. Außerdem war es notwendig, die Datensätze um ein zweites Postleitzahl-Feld für Postfächer zu erweitern. Beeinträchtigt werden konnten auch Teile des Formular- und Druckwesens, denn Geschäftspapiere, Visitenkarten, Kataloge, Publikationen, Stempel, Frankierautomaten etc. mussten rechtzeitig mit den neuen Adressen versehen werden [web7.5].

Die höchsten Kosten sind in den meisten Unternehmen bei der Programmierung angefallen. In zahlreichen Firmen mussten mehrere tausend Einzelanwendungen umgestellt werden. Dabei handelte es sich vorwiegend um Cobol-Programme, in denen zum Beispiel die Deklarationen solcher Variablen anzupassen waren, die die Postleitzahl enthielten. Auch die Datenstrukturen, Redefines und Renames konnten betroffen sein [web7.4]. Interne funktionale Abhängigkeiten von Variablen - zum Beispiel über Hilfsvariablen - wurden in Programmen häufig nicht gesehen.

7.4.3 Stellungnahme

Eine Änderung des Postleitzahlsystems war bei dem Entwurf von vielen Systemen nicht vorhersehbar. Daher könnte man den Entwicklern nur schwer vorwerfen, dass sie nicht mit der nötigen Voraussicht geplant hätten. Dennoch macht dieses Beispiel deutlich, wie wichtig es ist, ein System für Änderungen flexibel genug zu entwerfen. Weinberg sagt (zitiert nach [Sne91: 62]), dass es umso schwerer ist, ein System einer anderen Umgebung anzupassen, je mehr das System seiner spezifischen Umwelt angepasst ist. Systeme, die mit ihrer jeweiligen Umgebung nur lose gekoppelt sind, überleben am längsten. Bei den alten Postleitzahlen war es dagegen oft der Fall, dass die Programmlogik von dem Postleitzahlsystem abhängig gemacht wurde: In vielen Fällen wurden geografisch relevante Aussagen, etwa die Unterscheidung Ost- oder Westdeutschland, durch ein „O“ bzw. „W“ gekennzeichnet. Die vor

die Postleitzahl gesetzten Buchstaben sind ein Bestandteil der alten, nicht aber der neuen Postleitzahlen [web7.4]. Die alten Postleitzahlen wurden auch oft zu weiteren Ordnungskriterien benutzt: Eine postleitzahlbezogene Zuordnung von Vertriebsgebieten, von Tourenplanungen für Verkaufsfahrer und Techniker, von Umsatzstatistiken oder von Provisionsabrechnungen war besonders bei Grossunternehmen kein seltener Fall [web7.4]. Dies war mit den alten Postleitzahlen, die noch gebietsbezogen organisiert waren, kein Problem, aber die neuen fünfstelligen Postleitzahlen wurden in erster Linie nach funktionalen Kriterien erstellt. Programme, die Postleitzahlen nicht nur bearbeiteten, sondern diese als logische Codes benutzten, entsprechen nach Weinbergs Idee den Systemen, die ihrer spezifischen Umwelt angepasst sind. Er beschreibt weiterhin (zitiert nach [Sne91: 62]), wie sich Programme durch Testen und Benutzen immer mehr ihrer Hardware-Basis und ihren Eingabe-Daten anpassen.

> „Falls entweder die Daten oder die Hardware geändert wird, bricht die bisher reibungslos laufende Software zusammen. Falls sie durch genormte Schnittstellen von den Daten und der Hardware nicht ausreichend entkoppelt ist, wird sie auch nie wieder so gut funktionieren wie vorher. Sie bleibt nur als Krüppel mit verminderter Leistung am Leben" [Sne91: 62].

Die Erkenntnis aus den Änderungen der Programme wegen der Postleitzahlen ist, dass eine enge Abhängigkeit von diesen verhindert werden sollte. Das neue System der Post zog zwar neue Wartungstätigkeiten nach sich, aber eine optimale Wartung ermöglichte gleichzeitig eine Verbesserung der Software. Sneed zitiert dazu Basili und Turner:

> „Andererseits wird eine Software, die mehrere Datenstruktur- und Hardware-Veränderungen überlebt, zunehmend stärker und stabiler. Durch Evolution reift sie zum leistungsfähigen Produkt heran und stirbt erst, wenn ihre Funktionalität überholt ist" [Sne91: 62].

7.5 Beispielprojekt zum konzeptionellen Entwurf von Software-Systemen

Anhand eines Beispiels soll gezeigt werden, wie die Entwicklung zu erfolgen hat, damit Wartungsproblemen vorgebeugt werden kann. Es handelt sich dabei besonders um die Fälle der adaptiven, perfektiven und präventiven Wartung (vgl. 2.2). Während der Entwicklung soll dafür gesorgt werden, dass die Evolution der Software reibungslos erfolgen kann. Dazu sollen zuerst in 7.5.1 die wichtigsten Punkte zur konzeptionellen Entwicklung aufgelistet werden. Danach zeigt in 7.5.2 das Beispiel Universität, wie die Ideen in der Praxis umgesetzt werden können. Eine Auswertung dieses Beispiels erfolgt zum Schluss in 7.5.3.

7.5.1 Wie soll die Entwicklung stattfinden?

Es wurde bereits an mehreren Stellen erwähnt, dass mit späteren Änderungen eines IT-Systems gerechnet werden muss. Die damit verbundene Arbeit kann erleichtert und somit der Aufwand in der Wartung vermindert werden. Es handelt sich dabei weniger um die Fälle der korrektiven Wartung. Wie bereits in 3.2 dargestellt wurde, machen Fehlerkorrekturen nur einen geringen Anteil der Wartungstätigkeiten aus. Es gilt vielmehr, dass die adaptive und perfektive Wartung (vgl. 2.2) zum Teil erheblich erleichtert werden kann, wenn die Entstehungsgründe für diese schon während der Entwicklung konzeptionell erfasst werden. Durch Requirements Engineering ist eine so vollständige Anforderungsspezifikation zu erstellen wie nur möglich. Eine unterstützende Maßnahme ist, über den zu implementierenden Bereich hinauszuschauen. Die Systemgrenzen sollten weiter als die des unmittelbar DV-relevanten Implementierungs-bereichs gezogen sein: Die Umwelt, die in der Gegenwart außerhalb des betrachteten Implementierungsbereichs liegt, kann in der Zukunft bereits zu den nächsten Anforderungen gehören (siehe Abb. 7.1).

Abbildung 7.1

In Abbildung 7.1 liegt die Betonung auf der gegenwärtigen Situation. Dabei sind sowohl der Blick auf den Implementierungsbereich wie auch der Blick in die Umgebung (Konzipierungsbereich) wichtig. Reduziert man die Betrachtung nur auf das Notwendigste, so werden ausschließlich die Informationen erfasst, die später auch implementiert werden. Die zu enge Sicht führt aber in den meisten Fällen rasch zu Problemen, weil somit wichtige Zusammenhänge nicht erkannt werden können. [Holl99: 194] empfiehlt:

> „Man vermeide harte Systemgrenzen, sondern wähle einen weichen, verlaufenden Rand wie bei der Betrachtung durch eine kreisförmige Lupe mit stärkster Vergrößerung (höchster Genauigkeit) in der Mitte und nach außen hin abnehmender Vergrößerung (verminderter Genauigkeit). Man lässt die Genauigkeit in konzentrischen Ringen um das vermutete Kernproblem mit zunehmendem Durchmesser abnehmen" [Holl99: 194].

Eine weitere unterstützende Maßnahme ist, noch einen Aspekt zu erfassen: Es soll nicht nur die gegenwärtige Situation betrachtet, sondern auch die zukünftige Entwicklung erforscht werden. Durch Zukunftsprognosen können später eintreffende Sachverhalte eingeplant werden. Somit sind diese bereits in der ersten eingesetzten Version des IT-Systems konzeptionell berücksichtigt und folglich fällt die Realisierung der Erweiterungen (perfektive Wartung) leichter. Eine weitere Darstellung zur Abbildung 7.1 zeigt die bisher erwähnten Betrachtungen (siehe Abb. 7.2). Der Implementierungsbereich, den das IT-System unterstützen soll, ist in der Gegenwart und in der Zukunft abgebildet. Genauso sind auch die aktuelle Umgebung (Konzipierungsbereich) des Implementierungsbereichs und die zukünftigen visualisiert. In dem Zeitablauf werden neue Anforderungen implementiert und somit wächst der Implementierungsbereich in den Konzipierungsbereich hinein (siehe schwarz gekennzeichneter Bereich). Während der Weiterentwicklung sollte nicht nur der Implementierungsbereich wachsen, sondern auch eine größere Umgebung (Konzipierungsbereich) in der Betrachtung miterfasst werden.

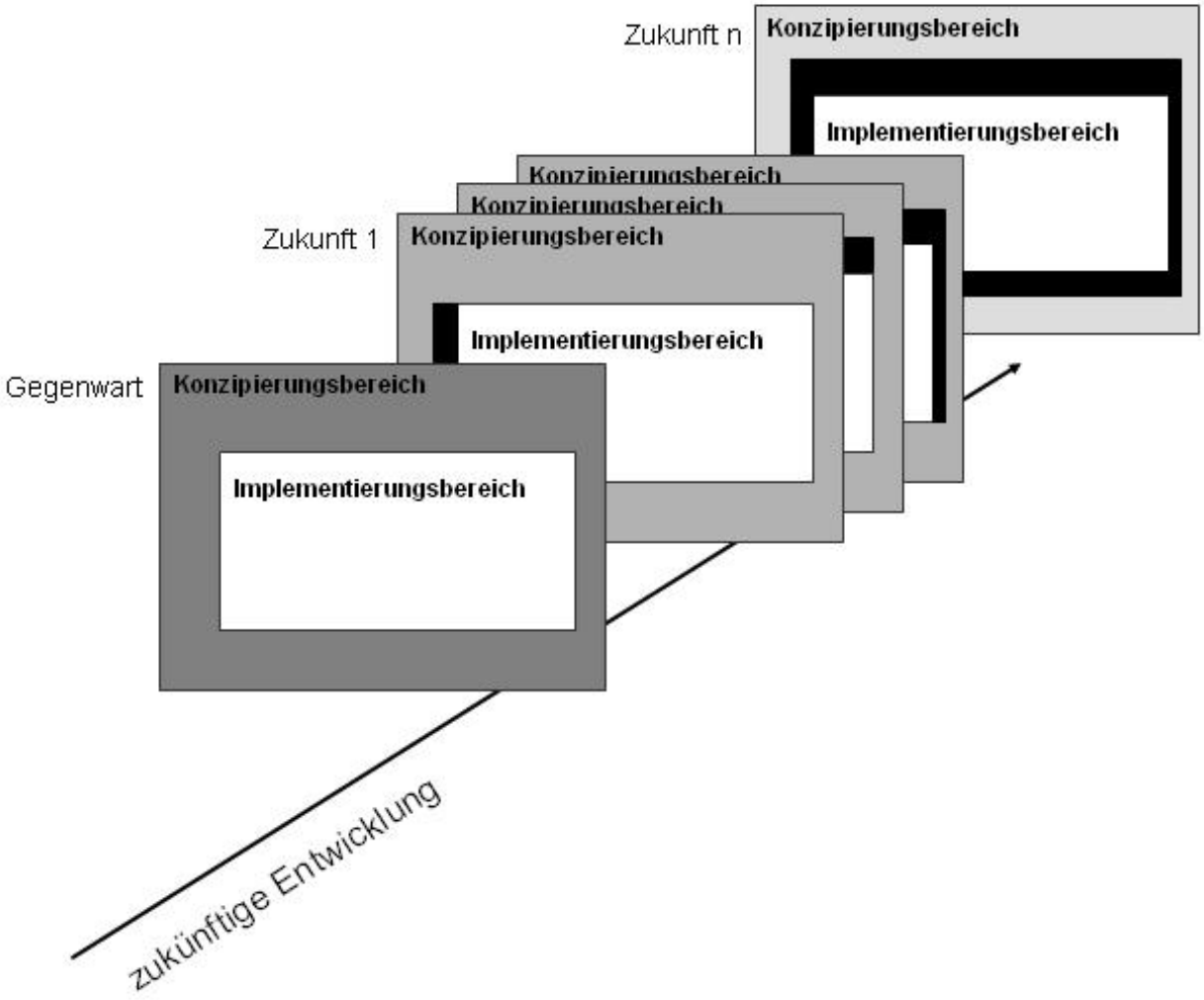


Abbildung 7.2

In Abbildung 7.2 sind folgende Aspekte erfasst:

a) Bei der Betrachtung des gegenwärtigen Implementierungsbereichs werden die nahe liegenden Anforderungen der aktuellen Situation gesammelt.

b) Der Blick in die Umgebung (Konzipierungsbereich) führt zu weiteren Erkenntnissen: Der Implementierungsbereich wird oft nur verständlich, wenn die Umgebung mitbetrachtet wird.

c) Mit Annahmen über den Implementierungsbereich der Zukunft können später eintreffende Aspekte erfasst werden.

d) Annahmen über die zukünftige Umgebung (Konzipierungsbereich) liefern Informationen über größere Zusammenhänge, die für die Implementierung wichtig sind.

Ziel eines Requirements Engineering ist es, Anforderungen möglichst für alle oben genannten Aspekte zu erfassen. Bis jetzt wurde der Blickwinkel bei vielen Entwicklungen auf ***a)*** reduziert. Tendenzen zur engen Systemabgrenzung sind darauf zurückzuführen, dass kleinere Struktureinheiten verglichen mit großen weniger komplex und besser überschaubar sind sowie effektiver bei Analogiebildungen [Holl99: 193]. Wenn aber Kenntnisse über den Randbereich hinaus nicht gewonnen werden, gehen wesentliche Zusammenhänge verloren. Die Betrachtung der Umgebung, wie in ***b)*** vorgeschlagen, führt dagegen zu einem besseren Ergebnis. Das folgende Beispiel soll dies verdeutlichen:

> „Zur Reduktion der Kapitalbindung im Rohstofflager unterstützt man den Einkauf mit EDV, übersieht dabei aber, dass eine Doppelbestellung von Rohstoffen durch Einkauf und Produktion erfolgt; das Problem liegt also woanders, ist aber wegen der zu engen Systemgrenzen nicht auszumachen" [Holl99: 193].

Wie in dem obigen Beispiel illustriert, können genauso Zusammenhänge, die anhand möglicher zukünftiger Entwicklungen (siehe ***c)*** und ***d)***) abgeleitet werden, zu wichtigen Erkenntnissen führen. Es kann sich dabei um vorhersehbare zukünftige Änderungen handeln, wie die Beispiele in 7.3.2 bzw. 7.3.3 gezeigt haben, oder um nicht vorhersehbare zukünftige Entwicklungen (siehe 7.3.4 und 7.3.5). Beim letzteren Fall können diese künftigen Entwicklungen nur indirekt berücksichtigt werden, indem das IT-System flexibel gehalten wird, d.h. Vorkehrungen für spätere Änderungen getroffen werden. Das IT-System soll so entworfen werden, dass spätere Änderungen leicht durchgeführt werden können. Das heißt, dass die Möglichkeit für spätere Änderungen schon vorweg eingeplant werden soll. Damit werden auch Vorkehrungen für die präventive Wartung (vgl. 2.2) getroffen. Künftige Anforderungen können so während der Software-Evolution in den späteren Systemversionen realisiert werden (vgl. 6.4.4).

7.5.2 Beispiel Universität

Anhand des folgenden Beispiels soll demonstriert werden, wie die bisherigen Erkenntnisse in der Entwicklung eingesetzt werden können. Es folgt eine Darstellung der Beziehung zwischen Dozenten und Vorlesungen. Daran sollen die Aspekte ***a)***, ***b)***, ***c)*** und ***d)*** aus 7.4.1 veranschaulicht werden. Durch ein konzeptionelles Datenmodell wird die Datenstruktur dieses Beispiels beschrieben.

Konzeptionelle Datenmodelle verwenden Konzepte wie Entitäten, Attribute und Beziehungen [Elm02: 46]. Eine Entität (Entity) stellt ein Modell eines Objekts oder Konzepts der realen Welt dar, z.B. ein Modell eines Angestellten oder eines Projekts, die in der Datenbank beschrieben werden. Ein Attribut stellt eine Eigenschaft dar, die die Beschreibung einer Entität weiter ausführt, z.B. Name oder Gehalt des Angestellten. Eine Beziehung (Relationship) zwischen zwei oder mehr Entitäten stellt einen Zusammenhang zwischen den Entitäten dar, z.B. eine Arbeitsbeziehung zwischen einem Mitarbeiter und einem Projekt. Das Entity-Relationship-Modell ist eines der beliebten konzeptionellen Datenmodelle, das auch für dieses Beispiel verwendet wird. Eine ausführliche Beschreibung dazu erfolgt hier nicht. Weiterführende Literatur findet man im Bereich Software-Engineering.

Das Beispiel Universität stellt die Beziehung zwischen Dozenten und Vorlesungen dar. Wird von der aktuellen Situation ausgegangen (Aspekt ***a)***), so gilt, dass ein Dozent zwar mehrere Vorlesungen halten, aber eine Vorlesung nur von einem einzigen Dozent gehalten werden kann. Für diese Situation ist eine Modellierung besonders einfach: Die Datenbank ist zunächst in zwei Tabellen organisiert. In der Tabelle Dozenten werden Daten über jeden Dozenten gespeichert, während die Tabelle Vorlesungen Daten zu allen Vorlesungen aufnimmt (siehe Abb. 7.3). Da für eine Vorlesung nur ein einziger Dozent in Frage kommt, kann durch den Fremdschlüssel DozentenID bestimmt werden, welcher zutreffend ist.

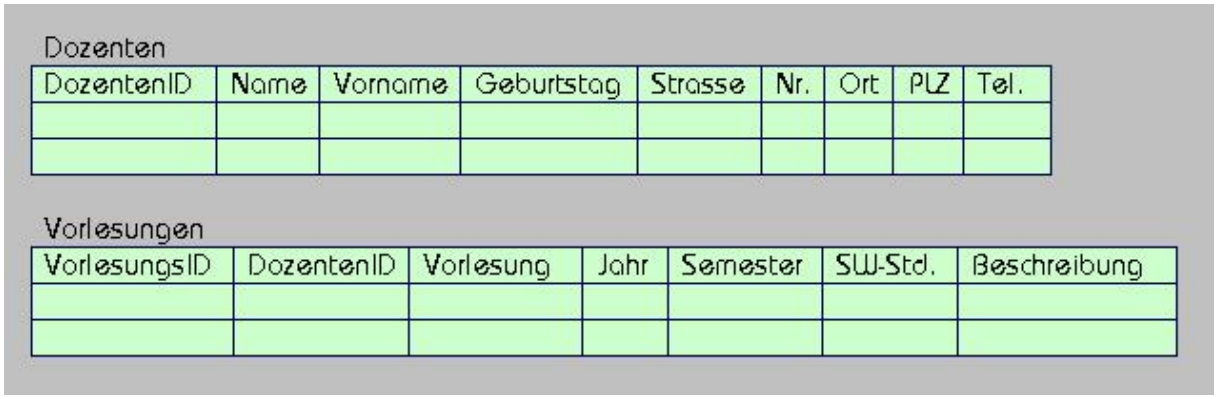


Abbildung 7.3

Der logische Zusammenhang wäre in der Datenbank wie folgt dargestellt (siehe Abb.7.4):

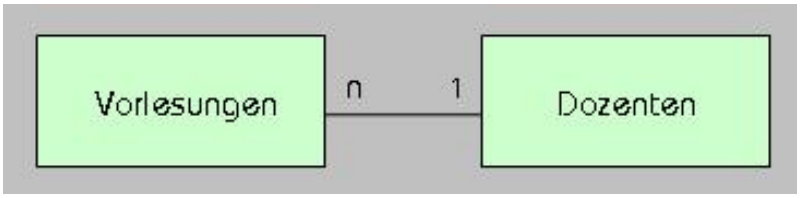


Abbildung 7.4

Dieses vereinfachte Modell zeigt, dass für eine Aufgabenstellung zwar eine korrekte Lösung angeboten werden kann, die aber dennoch Mängel aufweist. In diesem Beispiel würde eine Untersuchung der Umgebung (Aspekt ***b)***) zu der Erkenntnis führen, dass eine Modellierung mit nur zwei Tabellen bei späteren Änderungen zu Problemen führt. Ein Vergleich mit Abbildung 7.1 veranschaulicht, dass in dem obigen Modell nur der Implementierungsbereich berücksichtigt (Aspekt ***a)***), die Umgebung aber vernachlässigt wurde. Eine aktuelle Ausnahmesituation bei dem oben angenommenen Fall wäre, wenn eine Vorlesung nicht von einem, sondern von zwei oder mehreren Dozenten gehalten wird. Besonders bei Ausnahmesituationen der Gegenwart kann man darauf spekulieren, dass diese in der Zukunft zu Regelfällen gehören. Diese Ausnahmen sind nur bei einem erweiterten Blickfeld zu erkennen (vgl. Abb. 7.1) und sollten erforscht werden (Aspekt ***b)***). Wenn der mögliche Änderungsbedarf frühzeitig erkannt wird und somit die Ursachen für adaptive und perfektive Wartung (vgl. 2.2) vermindert werden, können spätere Korrekturmaßnahmen erleichtert oder ganz vermieden werden. In dem obigen Beispiel kann das Szenario, eine Vorlesung wird von zwei oder mehreren Dozenten gehalten, so dargestellt werden, dass die 1 : n - Beziehung durch eine m : n - Beziehung ersetzt wird (siehe Abb. 7.5):

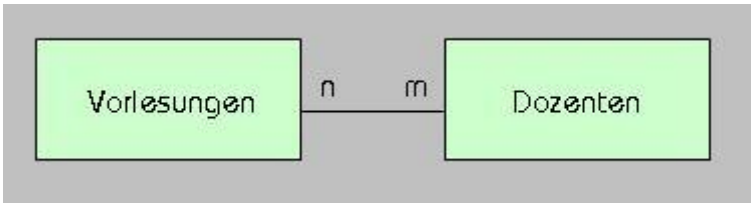


Abbildung 7.5

In der Datenbank ist diese Darstellungsform nicht direkt realisierbar. Durch die Aufnahme einer neuen Tabelle, die eine Zuordnung zwischen Dozenten und Kursen ermöglicht, ist die Beziehung wie folgt modellierbar (siehe Abb. 7.6):

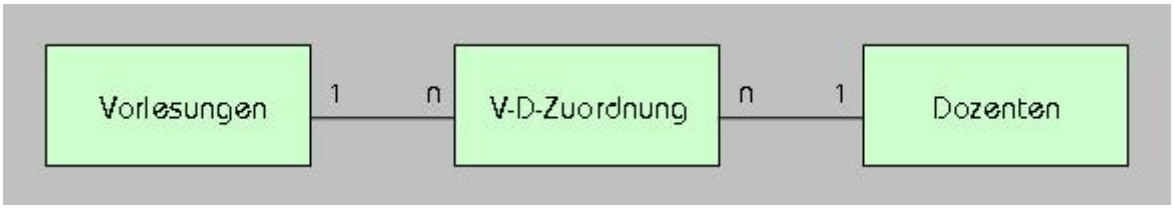


Abbildung 7.6

Bei dieser Darstellung ist die Tabelle der Dozenten, wie oben aufgeführt, eine sinnvolle Lösung, aber in der Tabelle der Vorlesungen ist der Fremdschüssel DozentenID überflüssig, weil dieser, wie auch der Fremdschüssel VorlesungsID, in der Zuordnungstabelle gespeichert wird. Abbildung 7.7 zeigt die drei Tabellen:

Dozenten

DozentenID	Name	Vorname	Geburtstag	Strasse	Nr.	Ort	PLZ	Tel.

Vorlesungen

VorlesungsID	Vorlesung	Jahr	Semester	SW-Std.	Beschreibung

V-D-Zuordnung

DozentenID	VorlesungsID

Abbildung 7.7

Dieses Modell ist auch für spätere Änderungen flexibel gehalten. Die gegenwärtigen Anforderungen sind ohne weiteres erfüllbar. D.h. Vorlesungen, die nur von einem Dozenten gehalten werden, können in der Datenbank problemlos gespeichert werden. Wenn aber später Vorlesungen stattfinden, für die mehrere Dozenten zuständig sind, können dafür die Daten auch aufgenommen werden, ohne die Notwendigkeit, Änderungen durchzuführen.

Weiterhin bietet das in Abbildung 7.6 aufgeführte Modell Raum für spätere Erweiterungen. Die Zuordnungstabelle kann z.B. spezifische Daten aufnehmen, die sich jeweils auf eine der Vorlesungs-Dozent-Kombinationen beziehen. Ein solche Anforderung könnte in der Zukunft (Aspekt ***c)*** bzw. ***d)***) entstehen. Eine sinnvolle Erweiterung hierzu ist die Speicherung der Start- und End-Daten. Damit kann leicht sichergestellt werden, dass ein Dozent für mehrere Vorlesungen nicht gleichzeitig eingeteilt wird. Abbildung 7.8 zeigt die Zuordnungstabelle mit

zu diesem Fall passenden Beispieldaten. Dabei sind in den ersten drei Zeilen Vorlesungen zu sehen, die in voneinander unabhängigem Zeitraum stattfinden. Die letzten zwei Zeilen dagegen zeigen zwei verschiedene Vorlesungen, die von demselben Dozent in der gleichen Zeit gehalten werden sollten.

V-D-Zuordnung

DozentenID	VorlesungsID	Jahr	Tag im Jahr	Start	Ende
12	56	2004	245	09:45	11:15
12	37	2004	245	11:30	13:00
12	42	2004	246	14:00	15:30
12	**45**	**2004**	**246**	**14:00**	**15:30**

Abbildung 7.8

Die in Abbildung 7.8 vorgelegte Situation stellt einen Fall dar, der in der Zukunft entstehen könnte. Anhand Abbildung 7.2 kann wieder verdeutlicht werden, dass hier über die gegenwärtige Situation hinausgegangen und bereits ein Teilbereich aus dem künftigen System bzw. dessen Umgebung (Aspekt ***c)*** bzw. ***d)***) konzipiert wurde.

Bei fortschreitender Untersuchung der gegenwärtigen Umgebung des Realitätsausschnitts könnten weitere Anforderungen (Aspekt ***b)***) erkannt werden. Z.B. könnte die Datenbank mehrere Fachbereiche übergreifen. Bei der jetzigen Darstellung handelt es sich nur um einen Fachbereich. Aus diesem Grund ist dieses gar nicht aufgeführt. Es könnte jedoch eine neue Tabelle für die Fachbereiche aufgenommen werden (siehe Abb. 7.9).

Fachbereiche

FachbereichsID	Fachbereich

Abbildung 7.9

Die einfachste Möglichkeit wäre dann, diese Daten in der Tabelle der Vorlesungen miteinzubeziehen. In Abbildung 7.10 ist die Tabelle mit dem Fremdschlüssel FachbereichsID erweitert worden. Über diesen kann ermittelt werden, zu welchem Fachbereich die jeweilige Vorlesung gehört.

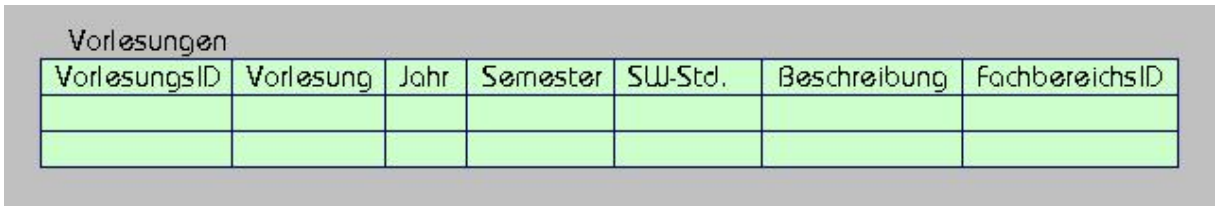


Abbildung 7.10

Allerdings ist diese Lösung nicht die allerbeste. Es wäre zwar für die gegenwärtige Situation ausreichend, aber ein Blick in die Zukunft (Aspekt ***c)***) kann zu neuen Erkenntnissen führen. Wenn einige Vorlesungen in mehreren Fachbereichen gehalten werden, so führt das in der Tabelle Vorlesungen sehr schnell zu unnötiger Redundanz. Abbildung 7.11 zeigt die zwei Tabellen mit den zu diesem Fall passenden Beispieldaten. Es ist leicht zu erkennen, dass die gleiche Vorlesung mehrfach gespeichert ist, wenn diese für mehrere Fachbereiche gehalten wird.

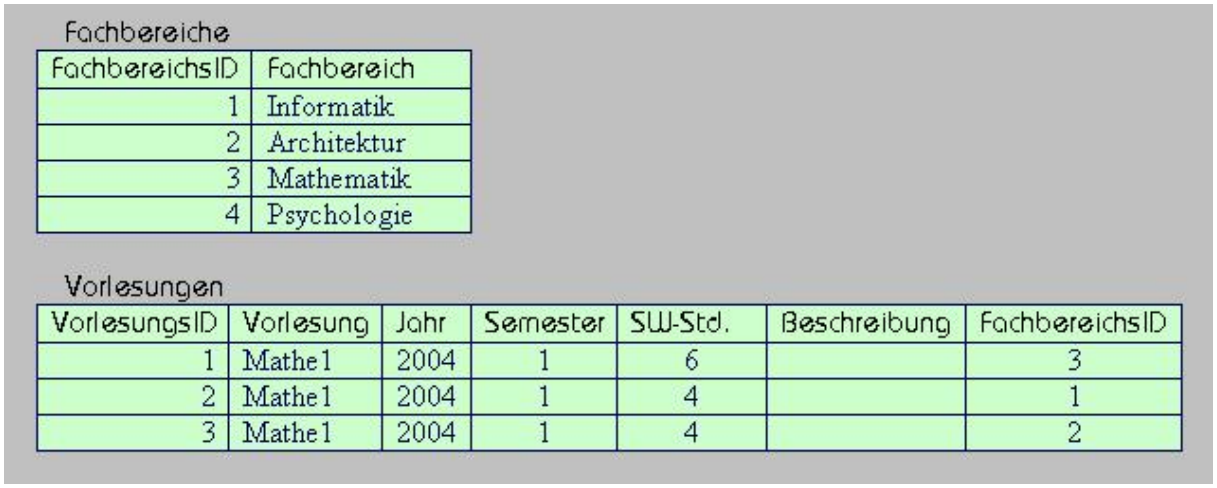


Abbildung 7.11

Eine bessere Lösung ist deswegen, wenn der Fremdschlüssel FachbereichsID in der Tabelle Vorlesungen nicht aufgenommen und der Fachbereich durch eine Zuordnungstabelle ermittelt wird. V-F-Zuordnung ist die Tabelle, die dann die Fremdschüssel FachbereichsID und VorlesungsID enthält (siehe Abb. 7.12).

Abbildung 7.12

Die Beziehungen zwischen den Tabellen können wie folgt modelliert werden (siehe Abb. 7.13):

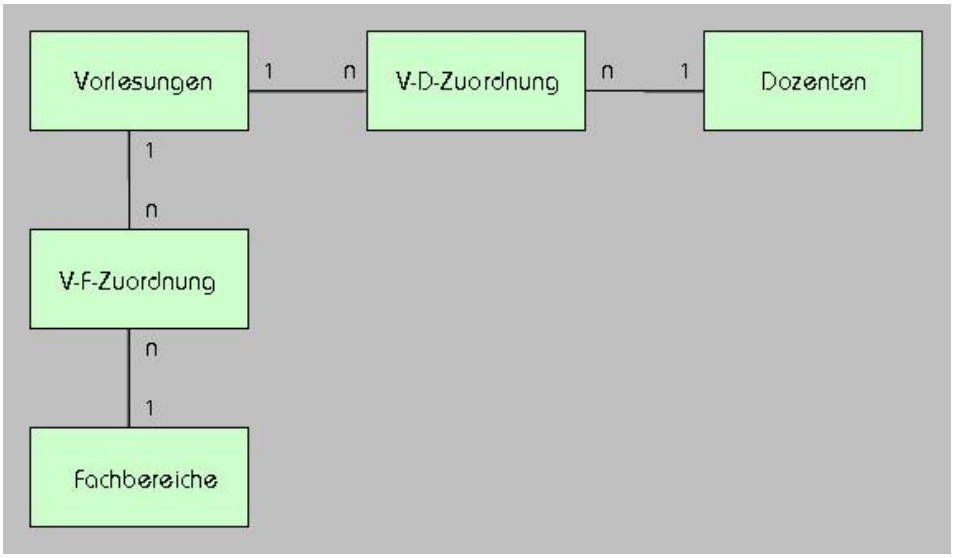


Abbildung 7.13

Dieses Beispiel kann noch weiter ausgearbeitet werden: In dem gegenwärtigen Modell wurde nicht festgelegt, wie historische Daten verarbeitet werden. Wenn nur die aktuellen Daten berücksichtigt werden, dann wäre die Betrachtung der Realität nur auf den Implementierungsbereich begrenzt (Aspekt ***a)***) (vgl. Abb. 7.2). Es wird aber davon ausgegangen, dass die Umgebung (Aspekt ***b*** bzw. ***d)***) in der Zukunft bald zu den relevanten Konzeptgrundlagen gehört. Es könnte sicherlich eine Zeitlang außer Acht gelassen werden, wie in diesem Beispiel mit historischen Daten umgegangen werden soll. In Abbildung 7.2 würden diese Daten nicht in dem Realitätsausschnitt, sondern in der Umgebung (Aspekt ***d)***) liegen. Aber früher oder später könnte das Problem, wie Vorlesungen vergangener Semester in dem Modell abgebildet werden, nicht mehr ignoriert werden. Parallel dazu können genauso Vorlesungen, die erst in der Planung sind, bzw. Vorlesungen der kommenden Semester betrachtet werden. D.h. die zukünftige Umgebung des Realitätsausschnitts (Aspekt ***d)***) (vgl. Abb. 7.2) sollte mitberücksichtigt werden. In der Tabelle Vorlesungen sind die Daten „Jahr“ und „Semester“ gespeichert (vgl. Abb. 7.7). Diese Felder ermöglichen zwar die Aufnahme von Daten für verschiedene Semester, aber somit müsste ein und dieselbe Vorlesung für jedes Semester separat gespeichert werden, was wiederum zu Redundanz führen würde. Sinnvoller ist die Lösung, wenn semesterabhängige bzw. –unabhängige Vorlesungsdaten voneinander getrennt verarbeitet werden. In Abbildung 7.7 ist die Tabelle Vorlesungen mit sieben Spalten zu sehen. Es könnten noch viele weitere Daten aufgezeichnet werden, aber um das Beispiel

überschaubar zu halten, wird darauf verzichtet. Abbildung 7.14 zeigt, wie die Daten auf zwei Tabellen aufgespaltet wurden: In der Tabelle Vorlesungshistorie befinden sich die semesterabhängigen und in der Tabelle Vorlesungsdaten die semesterunabhängigen Daten.

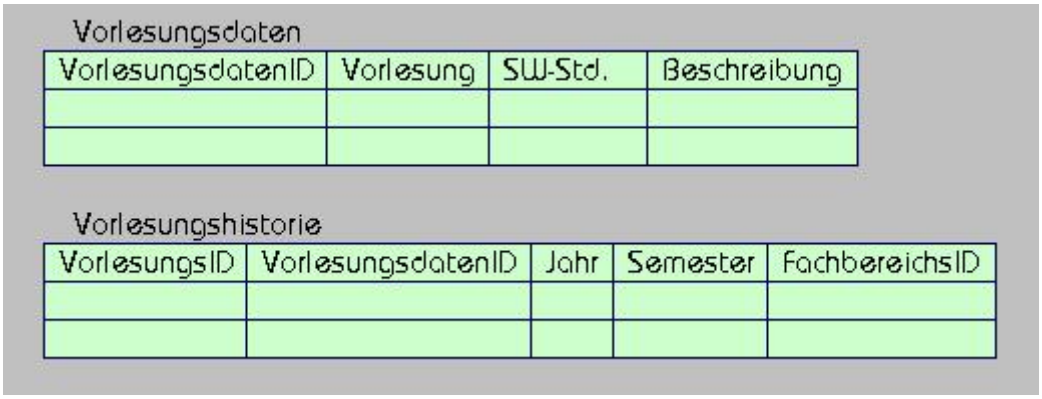


Abbildung 7.14

Abbildung 7.15 zeigt die Beziehungen zwischen allen Tabellen:

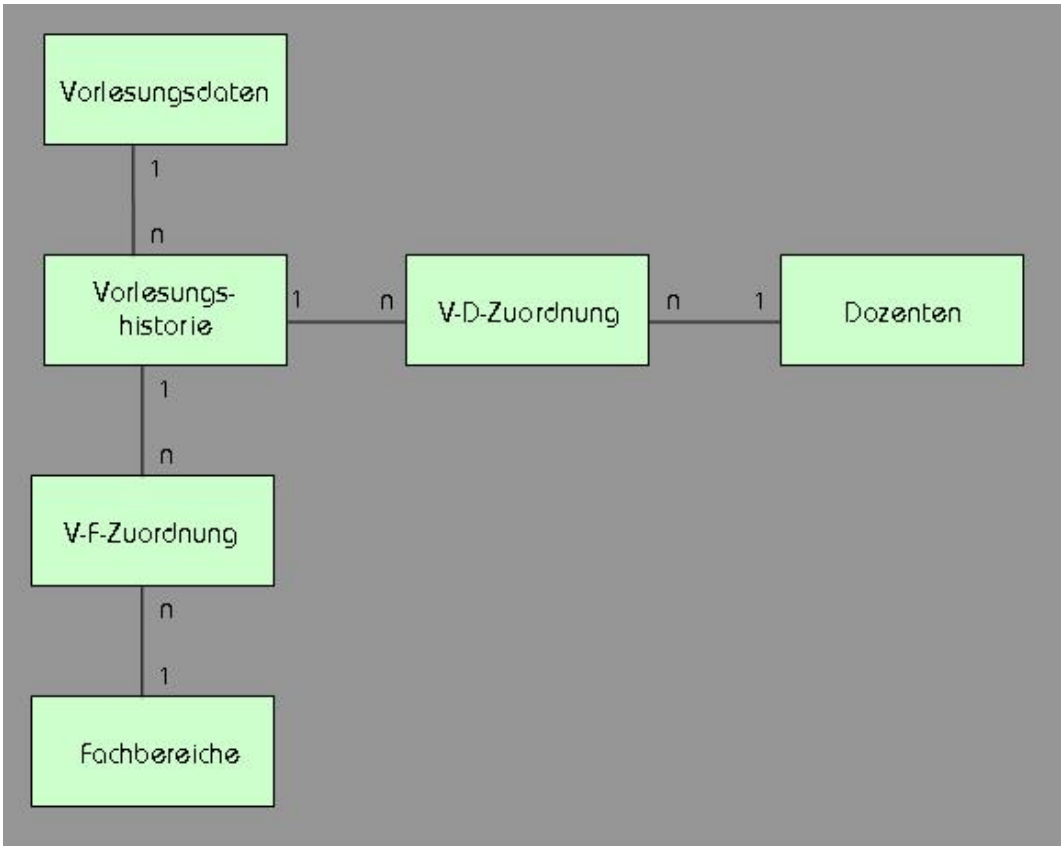


Abbildung 7.15

7.5.3 Zusammenfassung

Das Beispiel Universität ist ein aus dem Leben des Wirtschaftinformatikers gegriffenes Modell, das dazu dienen soll, bei einem Requirements Engineering mögliche Ideen zu präsentieren. Das Beispiel soll besonders den Weg zu einer zukunftsorientierten Modellierung aufzeichnen, wobei die Aspekte ***a)***, ***b)***, ***c)*** und ***d)*** aus 7.4.1 demonstriert wurden. Die Möglichkeiten für weitere sinnvolle Entwicklungen sind in 7.4.2 noch nicht ausgeschöpft. Die Erweiterungen dienen nur als Beispiele, die für weitere Fälle repräsentativ sind. Ziel dabei war zu zeigen, wie ein IT-System für spätere Änderungen flexibel gehalten bzw. für die Zukunft geplant werden kann.

8 Literaturhinweise

[Bal01] Balzert, H.: Lehrbuch der Software-Technik, Software-Entwicklung. Heidelberg; Berlin; Oxford: Spektrum 2001

Kommentar:

Typisches Buch aus dem Bereich der Software-Entwicklung, wo die typischen Phasen äußerst ausführlich (über 1000 Seiten) besprochen werden aber in dem das Kapitel Wartung kaum relevant auftaucht (10 Seiten). Somit hat mir dieses Buch nicht viel geholfen. Nur einen Absatz habe ich übernommen, der aber in dem Buch auch nur laut einem anderen Autor zitiert ist.

[Bit95] Bittner, U.; Hesse, W.; Schnath, J.: Praxis der Software-Entwicklung. München; Wien: Oldenbourg 1995

Kommentar:

Das Buch berichtet über das Projekt IPAS. Die Ergebnisse des Projektes sind Statistiken aus zahlreichen Untersuchungen. Die verschiedenen Statistiken decken alle Bereiche der Software-Entwicklung ab, sind gut strukturiert und kommentiert. Für diese Diplomarbeit sind sie dennoch nicht relevant. Nur durch einen Hinweis aus dem Buch habe ich meine Aussagen unterstützt.

[Bro97] Brockhaus, die Enzyklopädie. Leipzig; Mannheim: Brockhaus 1997

[Bru81] Brun, T.: Die Maintenance der Anwender-Programme: ein Sorgenkind? Management Zeitschrift io 50, 135-138. März 1981

Kommentar:

Ein guter Artikel über die Software-Wartung, der seine Aktualität seit 1981 nicht verloren hat. Der einfache Stil erleichtert das Lesen und macht die Themen auch für einen Laien verständlich. Er kann als Einführung für die Software-Wartung empfohlen werden, aber tiefere Einblicke in das Thema erlangt man nur durch andere Literatur. Somit habe ich aus dem Artikel nur einen Satz zitiert.

[Cur89] Curth, M. A.; Giebel, M. L.: Management der Software-Wartung. Stuttgart: Teubner 1989

Kommentar:

Ein empfehlenswertes Buch über die Software-Wartung, das mich in meiner Arbeit oft unterstützt hat. Es fasst die wichtigsten Aspekte der Software-Wartung zusammen, wobei die organisatorischen Aspekte stark hervorgehoben werden.

[Eis88] Eisner, P.: Strukturierte Software-Wartung. Dissertation der Rechts- und staatswissenschaftlichen Fakultät der Universität Zürich 1998

Kommentar:

Viele Themen werden in der Dissertation nur sehr kurz angesprochen. Die Struktur ist nicht klar und es gibt zu viele Überschriften, aber nur wenig dazugehörigen Inhalt. Dennoch ist die Dissertation als Einführung nicht schlecht, aber ohne zusätzliche Literatur nicht ausreichend. Einige Zitate habe ich daraus übernommen.

[Elm02] Elmasri, R.; Navathe, S. B.: Grundlagen der Datenbanksysteme. Kösel; Kempten: Addison-Wesley 2002

Kommentar:

Ein umfangreiches Buch über Datenbanksysteme, das auch als Nachschlagewerk geeignet ist. Mich hat es in meiner Arbeit bei dem Projektbeispiel Universität unterstützt. Das Buch ist klar gegliedert und die einzelnen Kapitel sind auch ohne Kontext verständlich.

[Fäs96] Fässler, J.: Eine syntaxbasierte Umgebung für Software-Wartungssysteme. Dissertation der Fachabteilung der Wirtschaftswissenschaftlichen Fakultät Zürich 1996

Kommentar:

Der erste Teil der Dissertation bietet einen Überblick über die Themen der Software-Wartung, der nicht besonders ausführlich ist. Deswegen war sie für meine Arbeit wenig förderlich. Der zweite Teil bezieht sich auf das Projekt AEMES, an dem Fässler gearbeitet hat. Dieser war für meine Arbeit nicht relevant.

[Hei87] Heinemann, K.: Software-Wartung. Münster: Lit 1987

Kommentar:

Das Buch eignet sich auch als Einführung in das Thema Software-Wartung. Der zweite Teil enthält viele Kenngrößen, Testberechnungen und Simulationen. Insgesamt geht es bei diesen Themen sehr mathematisch zu und deswegen hat mich besonders der erste Teil in meiner Arbeit unterstützt.

[Holl99] Holl, A.: Empirische Wirtschaftsinformatik und Erkenntnistheorie, in: Becker, Jörg [u.a.] (Hrsg.): Wirtschaftsinformatik und Wissenschaftstheorie. Wiesbaden: Gabler 1999

Kommentar:

Ein anspruchsvolles Buch, das sehr viele Informationen verdichtet. Ohne Vorkenntnisse ist es schwer zu lesen. Für meine Arbeit gab es einige wenige Themen, die an den bezeichneten Stellen zitiert sind.

[Kau94] Kaufmann, A.: Software-Reengineering. München; Wien: Oldenbourg 1994

Kommentar:

Eigentlich ist das Buch recht gut, weil es einfach zu lesen ist und die Themen anhand von Beispielen gut unterstützt werden. Für die Zwecke dieser Diplomarbeit ist es aber zu umfangreich und wurde nur in den entsprechenden Kapiteln zitiert.

[Kli99] Kliewer, R.: Reverse Engineering von Steuerungssoftware. Universität Kaiserlautern 1999

Kommentar:

Es handelt sich um eine Dissertation, die an vielen Stellen stark an die Lehrstuhlarbeit von Kliewer gebunden ist. Manche Themen sind dennoch allgemein gehalten und decken Bereiche aus Software-Wartung und Reengineering ab, die ich an einigen Stellen zitiert habe. Die Dissertation würde ich dennoch nicht weiterempfehlen, weil viele der Themen an anderer Stelle besser zu finden sind.

[Kro97] Kroha, P.: Softwaretechnologie. München: Prentice Hall 1997

Kommentar:

Ein Buch, welches das Thema Software-Wartung nur am Rande behandelt und deswegen für meine Arbeit nur wenig hilfreich war. Das Thema Reengineering wird auch angesprochen, aber in anderen Literaturen ist es besser behandelt.

[Leh91] Lehner, F.: Software-Wartung. München; Wien: Carl Hanser 1991

Kommentar:

Das Buch ist sehr einfach geschrieben, eine übersichtliche Gliederung macht das Lesen sehr angenehm. Es handelt eine breite Palette von Themen der Software-Wartung sowie Grundbegriffe ab. Somit hat mich das Buch in meiner Arbeit oft unterstützt und ich habe daraus viel zitiert.

[Leh99] Lehner, F. [u.a.] (Hrsg.): Software-Engineering II - Learning Textbook. Regensburg: Lehrstuhl für Witschaftsinformatik III 1999

Kommentar:

Dieses Buch ist sicherlich als Ergänzung zu der Lehrveranstaltung gedacht, ist

aber auch ohne diese verständlich, auch wenn dann das Material nicht ganz so ausführlich erscheint. Es ist nur als Zusatzliteratur zu anderen Büchern zu empfehlen. Aus dem Teil der Software-Wartung habe ich einiges zitiert.

[Lie80] Lientz, B. P.; Swanson, E. B.: Software Maintenance Management. London: Addison-Wesley 1980

Kommentar:

Das Buch ist das Ergebnis von Untersuchungen in 487 Unternehmen. Viele Statistiken und Zahlen machen die Themen sehr trocken, das Lesen fällt schwer. In dieser Diplomarbeit habe ich lediglich eine Abbildung übernommen.

[Mar83] Martin, J.; McClure, C.: Software Maintenance: The Problem and Its Solutions. Englewood Cliffs, New Jersey: Prentice-Hall 1983

Kommentar:

Auf das Buch wurde ich durch Zitate aus anderen Quellen aufmerksam. Ich habe dieses dennoch nicht oft benutzt und habe daraus hauptsächlich nur einige Abbildungen übernommen.

[Mül97] Müller, B.: Reengineering, Eine Einführung. Stuttgart: Teubner 1997

Kommentar:

Im Gegensatz zu dem Titel ist das Buch als Einführung weniger geeignet. In anderen Büchern wird man besser an das Thema herangeführt. Ich habe dieses Buch bis auf ein Paar Zitate kaum benutzt, weil die viele Graphen, Flussdiagramme, Kontrollflussdiagramme, Programmbeispiele für meine Diplomarbeit nicht gefragt waren.

[Sie03] Siedersleben, J.: Softwaretechnik, Praxiswissen für Software-Ingenieure. München [u.a.]: Hansen 2003

Kommentar:

In diesem Buch gab es, bis auf das Kapitel „Software-Renovierung“, keine Themen, die mich in meiner Arbeit unterstützt hätten. Nur wenige Zitate weisen auf das Buch hin.

[Sne91] Sneed, H.: Softwarewartung und –wiederverwendung, Bd I, Softwarewartung. Köln: Rudolf Müller GmbH 1991

Kommentar:

Ein sehr gutes Buch über die Software-Wartung, das mich in meiner Arbeit in vieler Hinsicht unterstützt hat, wie es aus den vielen Zitaten auch erkenntlich ist. Das Buch ist klar gegliedert, leicht zu verstehen und auch wenn es sich nicht

unbedingt einem spannenden Thema widmet, ist es nicht trocken geschrieben.

[Sne92] Sneed, H.: Softwarewartung und –wiederverwendung, Bd II, Softwaresanierung. Köln: Rudolf Müller GmbH 1992

Kommentar:

In diesem Buch wird Reverse- und Reengineerig sehr ausführlich vorgestellt. Das Buch ist ganz wie der erste Band: leicht verständlicher Stil, gute Erklärungen. Was dieses Thema anbelangt, hat mich dieses Buch am meisten unterstützt.

[Thu88] Thurner, R.: Technologie der Software-Wartung, in: Wix, B.; Balzert, H.: Softwarewartung. Mannheim; Wien; Zürich: Wissenschaftsverlag 1988

Kommentar:

Das Buch von Wix und Balzert enthält viele Aufsätze von anderen Autoren bezüglich der Software-Wartung. Das Thema wird in jedem Aufsatz neu eingeführt, nach anderen Aspekten besprochen und der Leser ist gezwungen sich mit dem Stil eines anderen Autors auseinanderzusetzen. Somit fällt die Lektüre insgesamt schwer und deswegen habe das Buch nur wenig benutzt.

9 Weiterführende Literatur

[Bal79] Balzer, R.: On the inevitable Interwining of Specification and Implementation. Comm. of ACM, Vol. 22, Nr. 5 Mai 1979

[Bal81] Balzer, R.: Transformational Implementation: An Example. IEEE Trans. On S.E., Vol. 7, Nr. 1 1981

[Bal82] Balzert, H.: Die Entwicklung von Software-Systemen. Mannheim; Wien; Zürich: Bibliographisches Institut 1982

[Bas90] Basili, V.: Viewing Maintenance as Reuse-Oriented Software Development. IEEE Software 1990

[Bis87] Bischoff, R.: Wartung und Pflege von Anwendungssystemen. HMD Nr. 135, 3-17 1987

[Boe76] Boehm, B. W.: Software Engineering. IEEE Transactions on Computers, Vol. C-25, No.12 , pp1226- 1241 Dezember 1976

[Bud80] Budde, R. Et. Al.: Untersuchungen über Maßnahmen zur Verbesserung der Softwareproduktion. München; Wien: GMD Bericht Nr. 130, Teil 1 1980

[Can72] Canning, R.: That Maintenance „Iceberg“. EDP Analyzer, Vol. 10, No. 10

Oktober 1972

[Clu93] McClure, C.: Software-Automatisierung: Reengineering - Repository – Wiederverwendbarkeit. München: Carl Hanser 1993

[Dat86] Ed: Tools for Software Structuring. Datamation Mai 1986

[Fis79] Fisher, D. A.: Standish Initial Thoughts on the Pebblemann Process. Institute for Defense Analyses January 3, 1979

[Gil88] Gilb, T.: Principles of Software-Engineering Management. Addison-Wesley 1988

[Hei90] Heinrich, L. J.; Burgholzer, P.: Informationsmanagement. München; Wien 1990

[Hos73] Hoskyns, J.; Co.: Implications ofUsing Modular Programming, Guide no. 1. 600 3rd Avenue, New York: Hoskyns Systems Research, Inc. 1973

[Inf80] Wallis, P. J.: Life-Cycle Mamagement. Maidenhead, England: Infotech State of the Art Report Series 8 No. 7, Volume 1: Analysis, Volume 2: Invited paper, Infotech International 1980

[JaL91] Jacobson, I.; Lindström, F.: Re-engineering of old systems to an objectoriented architecture. 1991

[Kau96] Kaufmann, A. H.: Software Reengineering. Darmstadt: TH, Fachbereich 1 Rechts- und Wirtschaftswissenschaften, Skript zur Vorlesung Version 1.3 Wintersemester 1996/97

[Leh69] Lehman, M.: The Programming Process. IYorktown Heights, NY: IBM Res. Rep. RC 2722 Sept. 1969

[Leh72] Lehman, M.; Belady, L.: An Introduction to Program Growth Dynamics, Statistical ComputerPerformance Evaluation. NY: W. Freiburger (ed.), Acad. Press 1972

[Lien80] Lientz, B. P.; Swanson, E. B.: Maintenance. 1980

[Raj97] Rajlich, V.: MSE: A Methodology for Software Evolution. Journal of Software Maintenance: Research and Practice 9, 103-124 1997

[Rig82] Riggs, R.: Computer Systems Maintenance. 1982

[Wat88] Waters, R. C.: Program Translation via Abstraction and Reimplementation. IEEE Trans. On S.E., Vol. 14, Nr. 8. August 1988

10 Quellen aus dem Internet

[web4.1] http://www.ifi.unizh.ch/swe/teaching/courses/evolution/papers/LehmanRamil97-Metrics-of-swevol.pdf

Kommentar:

Dieses Dokument handelt von Software-Evolution. Die Themen sind gut für Hintergrundwissen, waren aber für diese Arbeit nicht relevant. Somit habe ich es nur für ein Zitat verwendet: Gesetz der wachsenden Komplexität von Lehman.

[web6.1] http://www.doc.ic.ac.uk/~mml/seth_p2.pdf

Kommentar:

Insgesamt ist die Seite nicht so brauchbar für diese Diplomarbeit. Der erste Teil beschreibt die Entstehungsgeschichte von Software-Evolution, ist aber an anderer Stelle meist besser beschrieben. In meiner Arbeit ist nur ein Hinweis auf dieses Dokument zu finden.

[web6.2] http://www.ifi.unizh.ch/swe/teaching/courses/evolution/papers/SWEvol-1.pdf

Kommentar:

Das Skript hat nur stichwortartige Beschreibungen und vermittelt somit nur sehr wenig Kenntnisse. In meiner Arbeit habe ich daraus nur eine Definition für die Software-Evolution zitiert.

[web6.3] http://prog.vub.ac.be/FFSE/Meetings/39

Kommentar:

Das Skript ist, ohne den Vortrag gehört zu haben, kaum verständlich und vermittelt nur wenig brauchbare Kenntnisse. In meiner Arbeit habe ich nur die Definition für die Software- Evolution übernommen.

[web6.4] http://www.itee.uq.edu.au/~hearnden/_deltaware/ConfirmationReport.doc

Kommentar:

Dieses Dokument enthält zum Teil informatives Wissen über die Software-Evolution, aber andererseits werden viele zu spezifische Informationen über Architektur und anhand mathematischer Formeln dargelegte Erkenntnisse vermittelt, die ich in meiner Arbeit nicht gebrauchen konnte. Nur die Definition für die Software- Evolution habe ich daraus zitiert.

[web6.5] http://homepages.cwi.nl/~arie/rewiki/SoftwareEvolution.html

Kommentar:

Eine der wenigen Seiten im Internet, die Software-Evolution definiert, ohne das Bedürfnis lange Hintergrundwissen vermitteln zu wollen. Die Definitionen sind auch in meiner Arbeit zu finden. Darüber hinaus enthält die Seite keine sonstigen Informationen, nur Links zu anderen Webseiten, die ich nicht weiter verfolgt habe.

[web7.1] http://www-ea.inf.ethz.ch/Jahr2000/Download/Informatik.pdf

Kommentar:

Ein Dokument, welches über das Jahr-2000-Problem sehr ausführlich und leicht verständlich informiert. In dem Beispiel, in dem ich das Problem beschreibe, hat mich dieses Dokument sehr unterstützt.

[web7.2] http://agn-www.informatik.uni-hamburg.de/projects/archive/Jahr2000/arbeit/studienarb.html

Kommentar:

Diese Seite vermittelt umfangreiches Wissen über das Jahr-2000-Problem. In dem Zusammenhang werden mehrere Betriebssysteme, Programmiersprachen, Netzwerke, u.v.m. besprochen. Die Seite ist empfehlenswert. In meiner Arbeit habe ich diese nur für ein Beispiel benutzt.

[web7.3] http://www.gsp.com/2038/

Kommentar:

Die Seite enthält sehr wenige, aber spezifische, nützliche Informationen, an die ich in dem Beispiel von UNIX verwiesen habe.

[web7.4] http://www.computerwoche.de

COMPUTERWOCHE Nr. 22 vom 28.05.1993 Seite 7

Kommentar:

Der Artikel beschreibt sehr gut die Postleitzahl-Umstellung und ihre Auswirkung auf die IT-Welt. In meiner Arbeit ist dieses Ereignis auch beschrieben, wobei mir dieser Artikel sehr geholfen hat.

[web7.5] http://www.computerwoche.de

COMPUTERWOCHE Nr. 50 vom 11.12.1992

Kommentar:

Dieser Artikel bietet einen Einblick in die Problematik der Postleitzahl-Umstellung bezüglich der DV-Welt. Er ist etwas kurz, aber fasst die

wichtigsten Aspekte gut zusammen. In dem Beispiel, wo ich dieses Ereignis beschreibe, hat mich auch dieser Artikel unterstützt.

[web7.6] http://www.computerwoche.de
COMPUTERWOCHE Nr. 12 vom 19.03.1993 Seite 145-147
Kommentar:
Hierbei handelt es sich um einen Artikel, der die Postleitzahl-Umstellung besonders auf der technischen Ebene sehr gut beschreibt. Für das Beispiel der Postleitzahl-Umstellung hat mir dieser Artikel einige gute Aspekte geliefert.

11 Abbildungsverzeichnis

12 Anhang

Das Dokument [web4.1] enthält 113 Seiten. Dies ist der Auszug aus Seite 21:

No.	Brief Name	Law
I 1974	Continuing Change	*E*-type systems must be continually adapted else they become progressively less satisfactory.
II 1974	Increasing Complexity	As an *E*-type system evolves its complexity increases unless work is done to maintain or reduce it.
III 1974	Self Regulation	*E*-type system evolution process is self regulating with distribution of product and process measures close to normal.
IV 1980	Conservation of Organisational Stability (invariant work rate)	The average effective global activity rate in an evolving *E*-type system is invariant over product lifetime.
V 1980	Conservation of Familiarity	As an *E*-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour [leh80a] to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
VI 1980	Continuing Growth	The functional content of *E*-type systems must be continually increased to maintain user satisfaction over their lifetime.
VII 1996	Declining Quality	The quality of *E*-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
VIII 1996	Feedback System (first stated 1974, formalised as law 1996)	*E*-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Table 1 Laws of software evolution

21

Das Dokument [web6.1] enthält 7 Seiten. Dies ist der Auszug aus Seite 1:

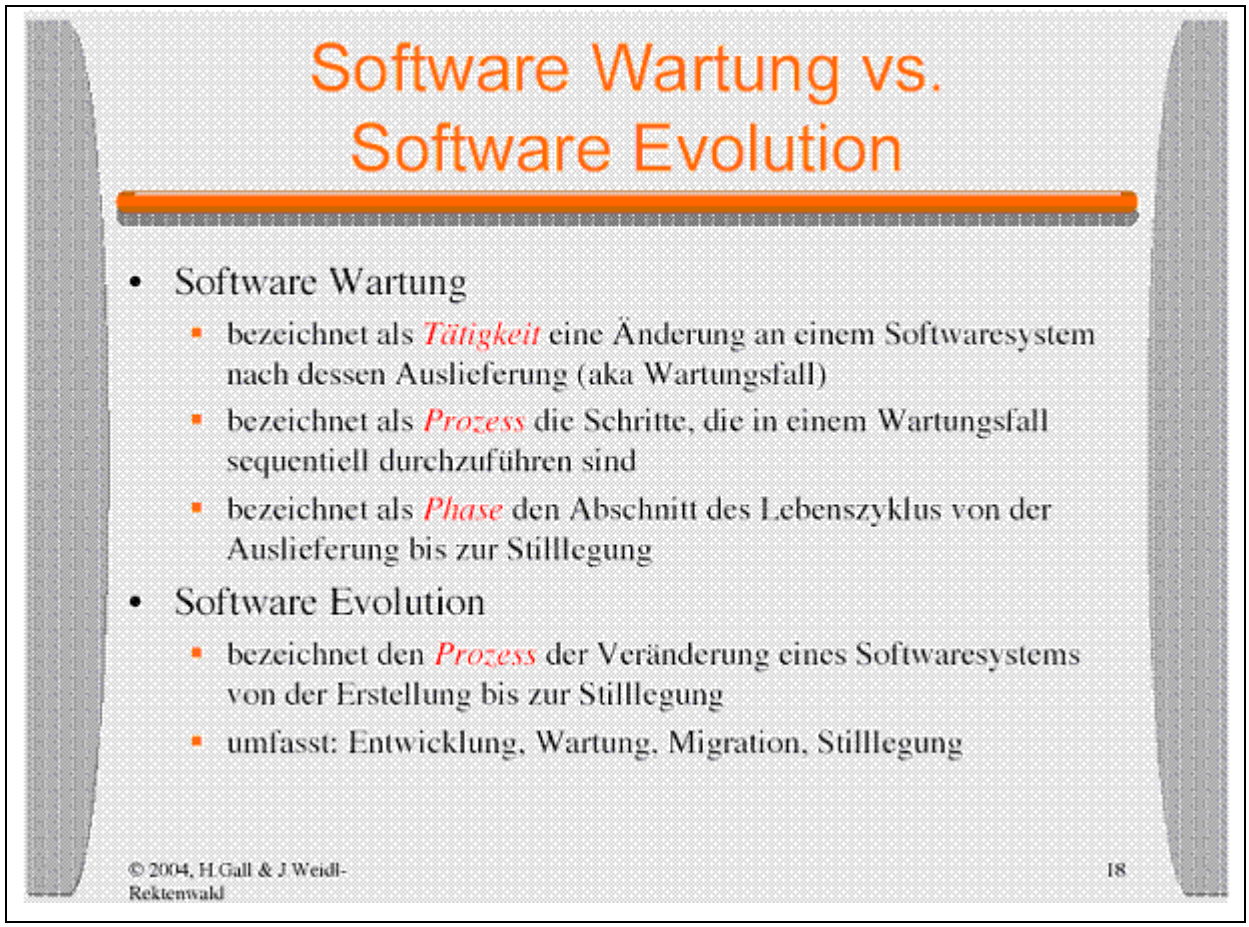
SETh - Approach to a Theory of Software Evolution
Case for Support Part 2: Proposed Research and Context

A Software Evolution

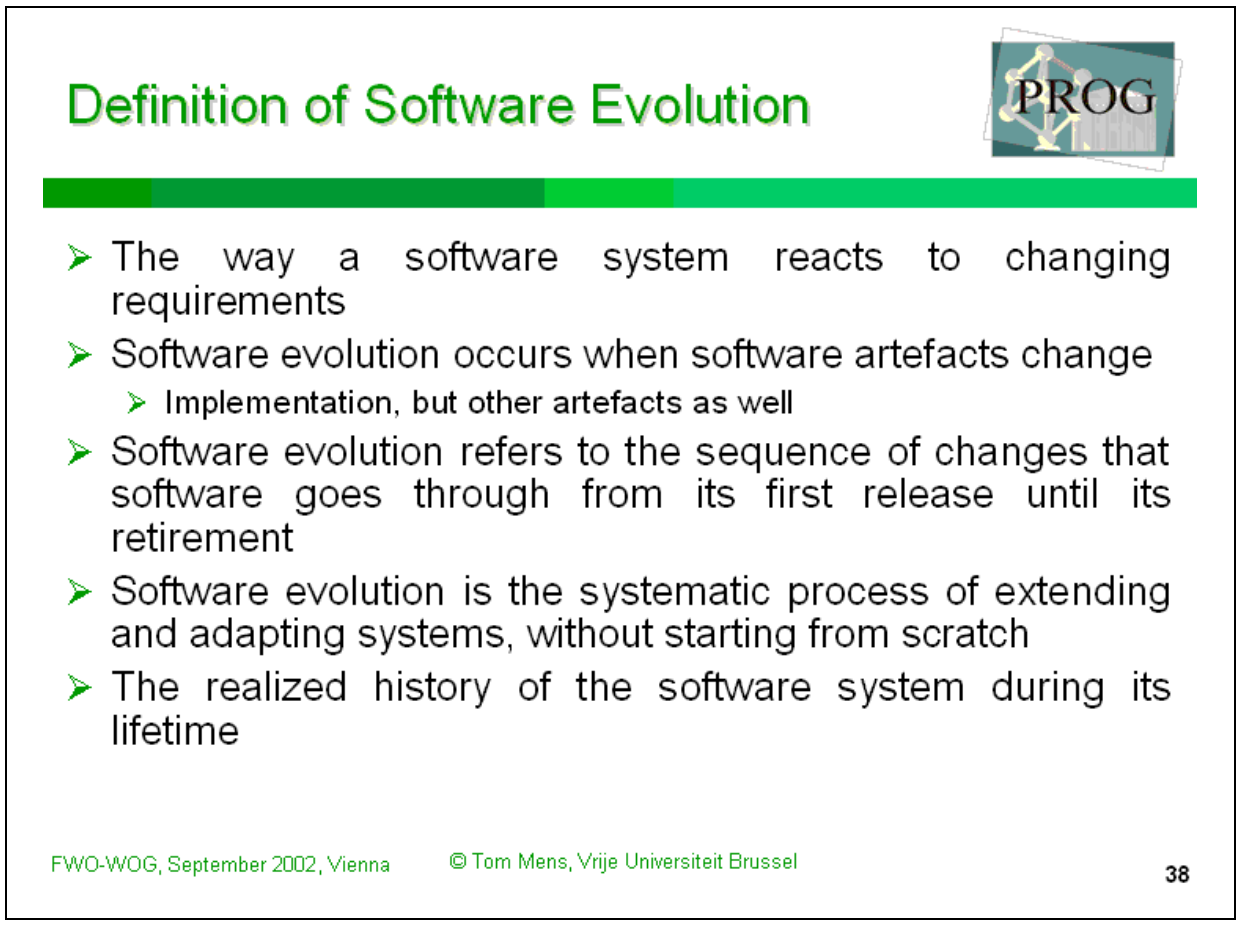
The phenomenon of software *evolution*, first identified in the early 70s [1,2], is now widely recognised as a topic worthy of serious investigation [3]. Studies undertaken have been largely *ad-hoc*, lacking a unifying conceptual framework, focussing mainly on the *how* of evolution [4] and on mechanisms for process improvement. The goal has been to increase productivity, reliability, adaptability and predictability of the software process, to decrease development time and to improve the quality of its product. Methods and tools making extensive use of formalisms and other mathematical techniques have been developed with the approach to their development and use primarily based on experience and intuition. The impetus for such investigation has come from the ubiquity of computers, and hence of software, in all aspects of human activity. Understandably, this has led to widespread, active, interest in *process improvement* in the context of software and, more generally, business processes as evidenced respectively by the SEI CMM model [5] and a recent EPSRC initiative [6]. A small number of groups have adopted a complementary approach [4], seeking to determine and understand the underlying *causes*, *attributes* and practical *impact* of evolution on the software process and its products. Their focus is on the *why* and *what* of software evolution; their goals, to address the practical nature of the phenomenon as experienced in industry and by users and to derive practical improvements. Such studies require observation of and data acquisition from industrial software processes and reliance on application of empirical methods [7]. Recently concluded, EPSRC supported, FEAST studies were based on this second approach and on a hypothesis that the software process is a complex, multi-level, multi-loop, multi-agent feedback system. The concepts underlying this and earlier work and their results have been widely disseminated [8].

[1]* Lehman MM, *The Programming Process*, IBM Res. Rep. RC 2722, IYorktown Heights, NY, Sept. 1969.
[2]* Belady LA and Lehman MM, *An Introduction to Program Growth Dynamics*, in Statistical Computer Performance Evaluation, W. Freiburger (ed.), Acad. Press, NY, 1972, pp. 503-511.

Das Dokument [web6.2] enthält 47 Seiten. Dies ist der Auszug aus Seite 18:



Das Dokument [web6.3] enthält 56 Seiten. Dies ist der Auszug aus Seite 38:



Das Dokument [web6.4] enthält 34 Seiten. Dies ist der Auszug aus Seite 24:

Rajlich [Raj97] describes evolution based on dependencies, and he defines 'software evolution' as

> "the changes in software requirements that result in consequent changes of the software."

[KTM+99] uses an alternative categorisation of maintenance activities, being either *corrections* or *enhancements*, where corrections resolve a discrepancy between requirements and behaviour, and enhancements modify the system by changing existing requirements, adding new requirements, or changing the implementation but not the requirements.

For the purposes of this thesis, a more useful definition is:

> "Software evolution is the process of making incremental change to software artefacts, while preserving the relationships between artefacts"

This definition constrains the *scope* of change (to being small in comparison with the artefacts being changed) as well as the *domain* of changeable entities (to being software artefacts). In this context, the term 'software artefact' means any piece of codified information that is *relevant* to the software product, where the relevance is captured by some form of (machine) interpretation. Thus implementation code is a software artefact. The software design is also a software artefact, if the refinement process uses it.

[KTM+99] Barbara Kitchenham, Huilherme Travassos, Anneliese von Maryhauser, Frank Niessink, Norman Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen and Hongji Yang, "Towards an Ontology of Software Maintenance", *Journal of Software Maintenance: Research and Practice* **11**, 365-389, 1999.

[Raj97] Rajlich, V, "MSE: A Methodology for Software Evolution", *Journal of Software Maintenance: Research and Practice* **9**, 103-124, 1997

Dies ist der Auszug aus der Webseite [web6.5]:

Software Evolution

Definitions

The Research Institute in Software Evolution defines software evolution as:

- *the set of activities, both technical and managerial, that ensures that software continues to meet organisational and business objectives in a cost effective way.*

Manny Lehman and Juan Ramil (2000) defined software evolution as:

- All programming activity that is intended to generate a new software version from an earlier operational version

Ned Chapin (1999) defines software evolution as:

- The application of SoftwareMaintenance activities and processes that generate a new operational software version with a changed customer-experienced functionality or properties from a prior operational version (...) together with the associated quality assurance activities and processes, and with the management of the activities and processes

According to the software life-cycle in the Software Maintenance And Evolution Roadmap, SoftwareEvolution is a particular phase in the SoftwareMaintenance process, immediately after initial delivery, but before servicing, phase out and close down.

Das Dokument [web7.1] enthält 6 Seiten. Dies ist der Auszug der Seiten 1, 2, 3 und 6:

Das Jahr-2000-Problem und die Informatik

Daniel Aebi

Das Jahr-2000-Problem (J2P), das gelegentlich auch als "Y2K-" oder "Millenium-Problem" bezeichnet wird, stellt nach übereinstimmender Meinung vieler Fachleute zur Zeit eine der grössten Herausforderungen für die Informatik dar. Obwohl die Problematik bereits seit einiger Zeit in verschiedenen Medien thematisiert wird, wird ihre Bedeutung mancherorts immer noch stark unterschätzt. Der folgende Beitrag gibt eine Uebersicht über die zugrundeliegenden Probleme und ihre Ursachen und erläutert mögliche Lösungsansätze.

Einleitung

In vielen Informatiksystemen werden Jahreszahlen in der auch im Alltag sehr gebräuchlichen, zweistelligen Kurzform – also beispielsweise "98" statt "1998" – repräsentiert, verarbeitet und gespeichert. Solche zweistelligen Darstellungen können beim Übergang vom Jahr 1999 auf das Jahr 2000 zu Problemen führen. So liefert beispielsweise die Berechnung des Alters einer im Jahre 1959 geborenen Person im Jahre 1997 (noch) ein korrektes Ergebnis ("97" – "59" = "38") dieselbe Berechnung führt aber im Jahr 2000 zu einem falschen Resultat ("00" – "59" = "-59"). Es treten aber nicht nur Schwierigkeiten bei Zeitraumberechnungen, sondern beispielsweise auch beim Sortieren und Mischen von Daten auf ("1997" < "2000" aber "00" < 97").

Selbst unter der Annahme, dass die Verarbeitung von Jahreszahlen – namentlich in kommerziellen Anwendungen – ein überaus häufiger Vorgang darstellt, ist man doch spontan geneigt, dieses Problem als vergleichsweise trivial zu betrachten.

Was führt nun dazu, dass das Jahr-2000-Problem von manchen als so überaus bedeutsam angesehen wird (siehe z.B. [Jones 97])? Zur Beantwortung dieser Frage wird im folgenden zuerst eine Klassierung von verschiedenen, in diesem Zusammenhang relevanten, Einzelproblemen vorgenommen.

Daniel Aebi studierte Wirtschaftsinformatik an der Universität Zürich. Seit 1985 ist er selbständig für kleine und mittlere Unternehmungen tätig. Von 1991 bis 1996 arbeitete er teilzeit als Assistent am Institut für Informationssysteme der ETH Zürich wo er im April 1996 mit der Dissertation "Re-Engineering und Migration betrieblicher Nutzdaten" promovierte. Seit Herbst 1996 arbeitet er teilzeit als Oberassistent an diesem Institut. Seine Interessengebiete umfassen Wartung, Betrieb und Migration von Anwendungssystemen.
e-mail: aebi@inf.ethz.ch

Was ist das Jahr-2000-Problem (J2P)?

Bei der Verarbeitung zeitbezogener Daten bieten nicht nur die eingangs geschilderten Überlaufprobleme, sondern auch die korrekte Behandlung von Spezialfällen – Schaltjahre, Bankwerktage, ... – oft Schwierigkeiten. Dazu kommt, dass Datum und Zeit nicht nur im Zusammenhang mit Anwendungsprogrammen und Daten, sondern natürlich auch bei systemnaher Software sowie bei Hardware und Firmware oft eine Rolle spielen.

Das Jahr-2000-Problem – im folgenden als J2P abgekürzt – stellt somit nicht ein *einzelnes* Problem dar. Vielmehr werden mit diesem Begriff eine ganze Reihe von Phänomenen umschrieben, die aber letztendlich alle einen Zeitbezug aufweisen. Deren grosse Heterogenität erschwert aber eine systematische Darstellung. Im folgenden wird versucht, die Probleme anhand von folgenden drei orthogonalen Kriterien zu gliedern:

- Problemart: "Was"
- Betroffene Systemkomponente: "Wo"
- Zeitpunkt des Auftretens: "Wann"

Beim Kriterium *Problemart* lassen sich die beiden Ausprägungen *Ueberlaufprobleme* sowie *Entwurfs- / Implementationsfehler* unterscheiden.

Ueberlaufprobleme: Damit werden Phänomene bezeichnet, die erst ab einem bestimmten Zeitpunkt zu Fehlfunktionen führen. Dazu gehört namentlich die eingangs geschilderte zweistellige Jahresrepräsentation.

Entwurfs- / Implementationsfehler: Dies umfasst Phänomene, die a priori falsch realisiert wurden, deren Fehlerhaftigkeit aber unter Umständen bisher nicht entdeckt wurde. Dazu gehören die bereits erwähnten Schaltjahr- oder Wochentagsberechnungen, aber beispielsweise auch die Mehrfachsemantik von Kalenderdaten (so werden zum Beispiel in manchen Anwendungen Daten wie der "9.9.99" oder der "11.11.11" als "Ablaufdatum unendlich", "kein Datenwert vorhanden" u.v.a.m. missbraucht). Auch die "Hartcodierung" von "19" zur Kennzeichnung des Jahrhunderts ist dazu zu zählen.

Zum Thema Schaltjahr ist zu bemerken, dass das Jahr 2000 aufgrund einer Regel mit *zwei* Ausnahmen – die in diesem konkreten Falle *beide* wirksam sind – ein Schaltjahr ist. Diese Tatsache ist aber nicht allgemein bekannt. So lassen sich leicht auch in modernen Programmierlehrbüchern falsche Algorithmen zur Schaltjahrberechnung finden[1].

Beim Kriterium *betroffene Systemkomponente* lassen sich *Datenbestände* (auf Sekundär- und Tertiärspeichern), *Hardware/Firmware*, *systemnahe Software*, *Individualsoftware*, *Standardsoftware* und *Erweiterungen* (z.B. auf Standardsoftware aufsetzende Anwendungen oder auch sog. "Enduser-Computing") unterscheiden.

Als drittes Kriterium schliesslich eignet sich der *Zeitpunkt des erstmaligen Auftretens* von Fehlfunktionen. Dabei ist es keineswegs so, dass nur gerade beim Übergang vom 31.12.1999 auf den 1.1.2000 Probleme auftreten. Manche Probleme treten schon viel eher auf.

Anhand dieser drei Kriterien lassen sich konkrete Einzelprobleme wie folgt einordnen (Fig. 1):

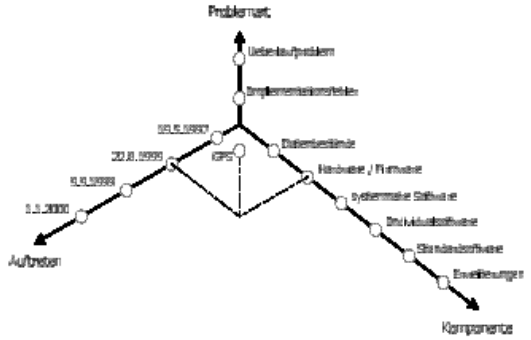


Fig. 1: Problemklassierung

Im folgenden soll anhand einiger ausgewählter Beispiele die "Breite" des J2P etwas veranschaulicht werden.

Datenbestände: "Date-in-keys", "embedded dates".
Kalenderdaten oder Teile davon bilden oft Bestandteile von anderen Datenfeldern und kommen auch häufig in Suchschlüsseln vor. Prominente Beispiele sind hier die AHV-Nummer oder die Studentenmatrikelnummer, die beide eine zweistellige Jahreszahl enthalten. Eine Erweiterung solcher Datenfelder auf vierstellige Repräsentation ist oft nicht so ohne weiteres möglich. Sortier- und Suchvorgänge stellen in diesem Zusammenhang besondere Schwierigkeiten dar.

Hardware/Firmware: PC-BIOS, GPS, Unterhaltungselektronik.
Viele Arbeitsplatzrechner verfügen über eine eigene interne Uhr. Bei einer grossen Zahl dieser Geräte (von denen viele auch noch nach dem 1.1.2000 im Einsatz stehen werden!) führt jedoch der Übergang vom 31.12.1999 auf den 1.1.2000 zu einer Rücksetzung des Systemdatums auf den 4.1.1980! Vergleichbare Probleme treten bei unterschiedlichsten Rechnern auf (nicht nur bei Arbeitsplatzrechnern).
Bei gewissen GPS-Empfängern (Global Positioning System) erfolgt am 22.8.1999 eine Rücksetzung eines Wochenzählers! GPS wird weltweit intensiv zur Navigation in Schiff- und Luftfahrt eingesetzt [GPS]. Die Konsequenzen dieses Überlaufes sind unklar (vielleicht sollte man aber am 22.8.99 besser nicht fliegen...).
Auch Geräte der Unterhaltungselektronik – beispielsweise Videorekorder – können betroffen sein. So sind heute Geräte im Handel, bei denen kein Datum nach dem 31.12.1999 eingestellt werden kann!

Systemnahe Software: OpenVMS, MS-Windows.
Am 19.5.1997 erfolgte in Betriebssystemen des Typs OpenVMS ein Überlauf eines Tageszählers (ein vergleichbares Phänomen ist in UNIX-Systemen am 19.1.2038 zu erwarten, siehe z.B. [UNIX-VMS]).
Der Dateimanager von MS-Windows 3.11 reagiert recht seltsam auf Daten nach dem 1.1.2000:

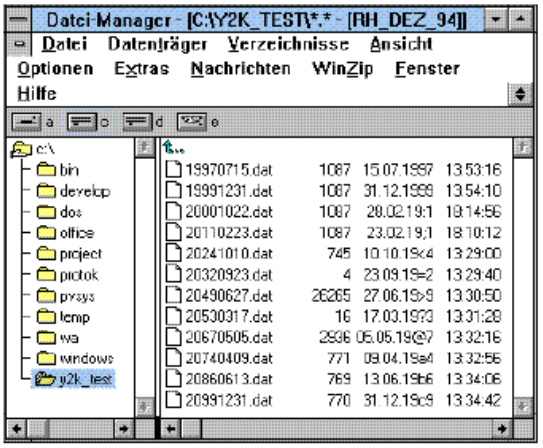


Fig. 2: MS-Windows 3.11-Dateimanager

Individualsoftware.
Für diese Kategorie lassen sich beliebig viele Beispiele finden. Empirische Untersuchungen haben gezeigt, dass in Programmen, die zeitbezogene Daten verarbeiten, etwa 3%-5% der Programmanweisungen mit Kalenderdaten zu tun haben und dass davon etwa ein Viertel angepasst werden müssen. In grösseren Umgebungen stellt das eine ganz enorme Herausforderung dar!

[1] Nebenbei bemerkt: Das dritte Jahrtausend beginnt erst am 1.1.2001 (eine Tatsache, die immer wieder zu intensiven, aber nutzlosen Diskussionen führt...).

Standardsoftware: SAP R/2, MS-Office-Produkte.
Eine Reihe von Herstellern hat – nicht zuletzt unter dem massiven Druck der Kunden – eingeräumt, dass bei Einsatz gewisser Releasestände ihrer Produkte mit Datumsproblemen zu rechnen ist (detaillierte Angaben sind z.B. unter [Compliance] zu finden).

Erweiterungen: Makrosprachen.
Werden Standardanwendungen durch eigenentwikkelte Komponenten (z.B. "Makros") ergänzt, so können auch mit solchen Komponenten Probleme auftreten.

Man ist mittlerweile auch zur Überzeugung gelangt, dass "embedded systems" (z.B. Haustechniksysteme, Systeme der Medizinaltechnik, Waffensysteme, ...) eine ganz erhebliche Problemquelle darstellen [Keogh 97].

Das J2P tritt also, entgegen oft gehörter Meinungen, nicht nur in sog. Legacy-Systemen im kommerziellen Umfeld auf. Vielmehr ist grundsätzlich bei jeder Art von System mit entsprechenden Problemen zu rechnen.

Ursachen

Aufgrund der im Einzelfall sofort einsehbaren möglichen Fehlfunktionen, stellt sich selbstverständlich die Frage nach den Ursachen. Wie konnte es überhaupt soweit kommen? Es sind eine ganze Reihe von Gründen, die zu dieser Situation geführt haben. Als eine wesentliche Ursache haben sicher *knappe Ressourcen* bei der Entwicklung (Speicherplatz, Rechenzeit, ...) eine Rolle gespielt. Dies bildet – in Kombination mit einer oft anzutreffenden Fehleinschätzung der Lebensdauer von Anwendungen – ein an sich einleuchtendes Argument. Das Einsparen von zwei Bytes hat aber schon seit längerem nicht mehr dieselbe Bedeutung und reicht als Rechtfertigung allein sicher nicht aus. Ein weiteres gewichtiges Argument ist in der *Mehrfachnutzung* von Programmteilen und Daten zu sehen. Namentlich mit dem Aufkommen relationaler Datenbanksysteme – die eine recht weitgehende Datenunabhängigkeit von Anwendungen unterstützen – war (und ist) es wirtschaftlich nicht vernünftig (und auch nicht gewünscht!), bestehende Datenbestände und darauf operierende Programme bei Einführung einer neuen Anwendung anzupassen. Dazu kommt, dass eine zweistellige Darstellung zu einer Vereinfachung von Ein-/Ausgabeschnittstellen führt (effizientere Dateneingabe, übersichtlichere Darstellung auf Masken und Listen u.a.m.).
Lange Zeit fehlende Standards zur Darstellung von Kalenderdaten (so wurde beispielsweise die ISO-Norm 8601 erst 1988 in Kraft gesetzt) haben zudem zu einer Vielzahl verschiedener Darstellungsarten von Kalenderdaten geführt.

Weitaus schwieriger fällt es allerdings, einleuchtende Gründe für entsprechende Sachverhalte im Bereich der systemnahen Software bzw. bei Hardware / Firmware zu finden. So ist es beim Besten Willen nicht einzusehen, warum ein Datentyp "Datum" in einem COBOL-Compiler nur zweistellige Jahreszahlen unterstützt [IBM97] oder warum sich die Uhr eines Videorekorders nicht auf ein Datum nach dem 31.12.1999 einstellen lässt...

Problemeigenschaften

Das J2P weist eine Reihe von interessanten Eigenschaften auf, die es von anderen bedeutsamen Wartungsvorhaben, die die Informatik bisher zu bewältigen hatte, unterscheidet. Die Informatik ist regelmässig mit komplexen Wartungsvorhaben konfrontiert. Als Beispiele sind etwa der Wechsel von der Warenumsatzsteuer zur Mehrwertsteuer in der Schweiz, die Umstellung des Postleitzahlensystems in Deutschland, oder die geplante Einführung des EURO zu nennen. Das J2P – das im Gegensatz zu den genannten Beispielen *adaptiver* Wartung ein rein *korrektives* Wartungsproblem ist – übertrifft all diese jedoch bei weitem. Folgende besonderen Eigenschaften zeichnen das J2P aus:

- *Betroffenheit.* Das J2P betrifft die ganze Informatik und zwar weltweit. Das Problem ist weder auf einzelne Anwendungstypen noch auf spezielle Systeme oder einzelne Länder beschränkt.
- *Kein Handlungsspielraum.* Wer die Probleme nicht löst, muss mit Fehlfunktionen rechnen.
- *Fixe deadline.* Das Problem muss unter hohem Zeitdruck, *unbedingt termingerecht* gelöst werden. Die Informatik hat jedoch erfahrungsgemäss gelegentlich etwas Mühe mit dem Einhalten von Terminen ...
- *Kein Mehrwert.* Vordergründig entsteht durch die Behebung des Problems kein Mehrwert. Das führt dazu, dass die Versuchung sehr gross ist, entsprechende Sanierungsarbeiten mit anderen – evtl. längst geplanten – Erweiterungen zu kombinieren, was oft zu einer Erhöhung des Projektrisikos führt.
- *Kein "problem-owner".* Da es sich bei J2P-Sanierungsprojekten um Querschnittsprojekte handelt, fühlt sich niemand so richtig zuständig.
- *Hohe Kosten, schwierig.* Die Lösung des J2P ist sehr komplex und teuer. Entsprechende Sanierungsprojekte wurden aber nicht langfristig geplant, die benötigten Ressourcen (Personal, Geräte, Räume, ...) sind dementsprechend oft nur schwierig bereitzustellen.

Aufgrund dieser besonderen Eigenheiten des J2P erstaunt es auch nicht weiter, dass bisher nur recht wenig Erfahrung im Umgang mit Projekten dieser Art vorhanden ist. Es wurden mittlerweile zwar eine

Literatur

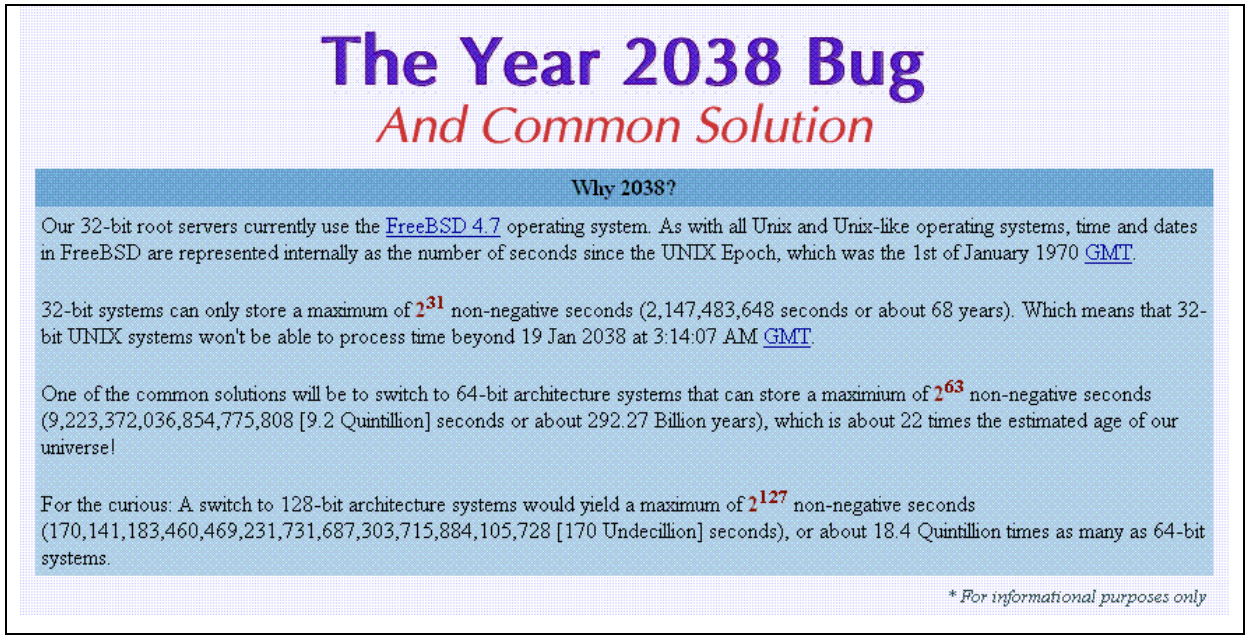
[Aebi 96]
Aebi, D.: Re-Engineering und Migration betrieblicher Nutzdaten. Dissertation Nr. 11628. ETH Zürich. 1996.
ftp://ftp.inf.ethz.ch/pub/publications/dissertations/th11628.ps
[Aebi 97a]
D. Aebi: Wer das Problem verdrängt, stürzt ab. In: io Management, 4/1997.
[Aebi 97b]
D. Aebi et al.: Unterlagen zum Industriekurs “Das Jahr 2000 und die Informatik”. 15.7.1997, ETH Zürich.
[Compliance]
http://www.wa.gov/dis/2000/6_survey.htm
[de Jager, Burgeon 97]
Peter de Jager, Richard Burgeon: Managing OO. Surviving the Year 2000 Computing Crisis.Wiley, 1997.
[Estes 97]
D. Estes: Encapsulation Solutions For Year 2000 Compliance. Working Paper. 1997.
http://www.jks.co.uk/y2ki/recent/pending/y2kenca4.zip
[GPS]
Fehlerbeschreibung: http://www.stl.nps.navy.mil/lists/c4i-pro/5215.html
Allg. Informationen: http://www.utexas.edu/depts/grg/gcraft/notes/gps/gps.html
bzw. http://www.navcen.uscg.mil
[IBM 97]
IBM: The Year 2000 and 2-Digit Dates: A Guide for Planning and Implementation. 6th Edition. 1997.
http://www1.s390.ibm.com/stories/year2000/downloads/y2kpaper.ps
[Jones 97]
Capers Jones: The Global Economic Impact of the Year 2000 Software Problem.
ftp://ftp.spr.com/articles/y2k52.pdf
[Keogh 97]
J. Keogh: Solving the Year 2000 Problem. Academic Press. 1997.
[Ragland 97]
Bryce Ragland: Year 2000 Problerm Solver. Mc Graw Hill, 1997.
[Ulrich, Hayes 97]
W. Ulrich, I. Hayes: The Year 2000 Software Crisis. Prentice Hall. 1997.
[UNIXVMS]
http://www.sun.com/solaris/Interactive/interactive_y2000.html
http://www.openvms.digital.com/openvms/products/year-2000/index.html
[Wagner 97]
R. A. Wagner: Solving the Date Crisis. Communications of the ACM. Vol. 40. No 5. May 1997.

In der Webseite [web7.2] sind 6 Kapitel aufgeführt. Dies ist der Auszug aus dem Dritten:

3.1.1.4 UNIX

UNIX ist ein Betriebssystem, daß von den Entwicklern mit der nötigen Weitsicht geplant wurde. So wird UNIX meist auf Computern mit 32bit Architektur eingesetzt. Aus diesem Grund verwaltet UNIX das Datum in Form einer 32bit Integer Zahl. Diese Art der Datumsinterpretation nennt man auch serielles Datum. In einer Integer-Variablen werden die Sekunden seit dem 01.01.1970, der Geburtsstunde von UNIX, fortgeschrieben. Ein Überlauf dieser Variablen findet am 19.01.2038 um 3:14:07 Uhr statt, denn dann ist der maximale Wert, den diese Variable speichern kann erreicht. UNIX wird also den Jahr-2000-Wechsel problemlos überstehen. Das Betriebssystem liefert das Datum den System- und Anwenderprogrammen über eine Vielzahl von API-Funktionen (***A**pplication **P**rogramming **I**nterface*), sowie Verwalter- und Benutzerbefehle. Die API-Schnittstelle erlaubt es, mit gezielten Befehlen das Datum und die Uhrzeit abzufragen und/oder zu beeinflussen. So existieren einige Funktionen, die es durch zusätzliche Parameter erlauben, zwischen zwei- und vierstelliger Jahreszahlinterpretation sowohl in der Darstellung, als auch in der Verarbeitung zu unterscheiden. Hier gilt es, die Systemsoftware auf eventuell falsche Funktionsaufrufe zu überprüfen.

Dies ist der Auszug aus der Webseite [web7.3]:



The Year 2038 Bug

And Common Solution

Why 2038?

Our 32-bit root servers currently use the FreeBSD 4.7 operating system. As with all Unix and Unix-like operating systems, time and dates in FreeBSD are represented internally as the number of seconds since the UNIX Epoch, which was the 1st of January 1970 GMT.

32-bit systems can only store a maximum of 2^{31} non-negative seconds (2,147,483,648 seconds or about 68 years). Which means that 32-bit UNIX systems won't be able to process time beyond 19 Jan 2038 at 3:14:07 AM GMT.

One of the common solutions will be to switch to 64-bit architecture systems that can store a maximum of 2^{63} non-negative seconds (9,223,372,036,854,775,808 [9.2 Quintillion] seconds or about 292.27 Billion years), which is about 22 times the estimated age of our universe!

For the curious: A switch to 128-bit architecture systems would yield a maximum of 2^{127} non-negative seconds (170,141,183,460,469,231,731,687,303,715,884,105,728 [170 Undecillion] seconds), or about 18.4 Quintillion times as many as 64-bit systems.

** For informational purposes only*

Dies ist der Auszug aus der Webseite [web7.4]:

Die Postleitzahl-Umstellung blockiert viele DV-Projekte Auswirkungen der Wirtschaftskrise werden noch verstaerkt

COMPUTERWOCHE Nr. 22 vom 28.05.1993 Seite 7

Die Datenverarbeitung als Rationalisierungsinstrument steht in Zeiten konjunktureller Talfahrt vor ihrer wohl groessten Herausforderung. Hinter Management-Strategien wie Business Re- Engineering, schlanker Fertigung oder Abflachung der unternehmensinternen Hierarchien verbergen sich neue DV-Konzepte, deren Realisierung in den Verantwortungsbereich der IT-Chefs faellt. Die aber muessen sich zur Zeit mit einem anderen Problem herumschlagen: Vom 1. Juli 1993 an gelten in der Bundesrepublik neue Postleitzahlen. Saemtliche Adressbestaende und Anwendungen sind mit hohem Kostenaufwand anzupassen. Die deutsche Wirtschaft hat auf die Postleitzahlen-Umstellung nicht gewartet. Sie muesste eigentlich ganz andere Probleme bewaeltigen." Dieter Struve, Systemanalytiker und Umstellungsexperte beim Frankfurter Neckermann Versand, bringt auf den Punkt, was viele DV-Verantwortliche denken: Von der Postleitzahl-Reform haben Anwender keinerlei Vorteile zu erwarten. In einer Reihe von Konzernen stuenden zur Zeit eigentlich andere Dinge an: Die Dezentralisierung der DV-Strukturen, die Verteilung von Daten und Anwendungen ueber verschiedene Plattformen hinweg sowie die Einfuehrung leistungsfaehiger Client-Server-Architekturen. Doch die Post hat mit ihrer Reform Sachzwaenge geschaffen, an denen kein Unternehmen vorbeikommt. Begruendet wird die Einfuehrung der fuenfstelligen Postleitzahlen mit der Vereinigung Deutschlands. Insgesamt 800 Postleitzahlen in West- und Ostdeutschland, so argumentiert die Behoerde, seien doppelt vorhanden. Man wolle daher ein einheitliches und eindeutiges System schaffen. Bei der Gelegenheit lasse sich auch das ueberholte Briefverteil-Verfahren rationalisieren. Die Dimension der Umstellungsarbeiten ist nicht leicht abzuschaetzen, viele Unternehmen tun sich mit der Kalkulation der Kosten schwer. So beziffert beispielsweise die Techniker Krankenkasse in Hamburg den Aufwand fuer die Bearbeitung ihres kompletten Adressbestandes von etwa sieben Millionen Anschriften mit rund zwei Millionen Mark. Siegfried Laskawy, Leiter des Megaprojektes, betont aber, dass es sich dabei zunaechst nur um geschaetzte "Nettokosten" handelt. Seiner Ansicht nach kommt noch eine nicht genau zu kalkulierende Summe fuer interne Verbindungen von Projekten und deren Auswirkungen hinzu. In der Abteilung Software-

Entwicklung liegen bei der Krankenkasse einige wichtige Projekte seit nunmehr gut einem halben Jahr auf Eis. "Draussen warten die Geschaeftsstellen darauf, dass wir unsere neuen Softwaresysteme einfuehren - das geht aber nicht, weil die Umstellung der Postleitzahlen Vorrang hat. Dafuer ist in einigen Sachgebieten quasi alles andere niedergelegt worden", bilanziert der Projektkoordinator. Mit dem 1. Juli, so Laskawy, sind die Probleme noch nicht vom Tisch. Zwar duerfte dann das Gros der vorhandenen Adressen ersetzt worden sein, doch eine Reihe heute kaum kalkulierbarer Probleme werden wohl erst zum Umstellungszeitpunkt sichtbar. Durch aufwendige Simulationen - drei sind noch im Juni geplant - versucht Laskawy die Probleme vorher aufzudecken und entgegenzuwirken. Die Techniker Krankenkasse rechnet damit, etwa 70 000 Adressen manuell nachbearbeiten zu muessen. Diese Quote von voraussichtlich gut einem Prozent kann sich sehen lassen, in anderen Konzernen werden Werte zwischen fuenf und 15 Prozent erreicht. Das guenstige Ergebnis dem Umstand zu verdanken, dass das Unternehmen seit Jahren ein selbstgeschriebenes Adresspruefsystem einsetzt. Dennoch ist fuer die Versicherung ein Bodensatz an nichtkonvertierbaren Adressen unvermeidlich. Dafuer sorgen schon die fehlerhaften Leitdateien der Bundespost, die eine komplette Eins-zu-eins-Uebertragung praktisch ausschliessen. "Es gibt Strassen, die in den Dateien der Post gar nicht auftauchen", nennt Laskawy eine der wichtigsten Fehlerquellen, das Strassenverzeichnis. Besonders die ostdeutschen Strassen seien nicht vollstaendig aufgefuehrt. Die Post entschuldigt diese Luecken mit der Umbenennung zahlreicher Strassen in Ostdeutschland. Volkshelden wie Walter Ulbricht, Ernst Thaelmann oder Rosa Luxemburg hatten mit der Wiedervereinigung ausgedient und mussten nach und nach von den Strassenschildern weichen. Dadurch ist das Verzeichnis nicht komplett, eine Reihe von Anschriften laesst sich nicht den einzelnen Postleitzahlbezirken zuordnen. Eine schwierige Situation, so Laskawy, ergibt sich auch in Berlin, wo vor etwa zwei Jahren die Menge der Postzustellaemter reduziert worden war. Diese Reform habe aber nur auf dem Papier stattgefunden. So gebe es heute fuer eine Reihe von Postzustellbezirken in der Bundeshauptstadt zwei gueltige vierstellige Postleitzahlen: eine amtliche, die postintern offiziell benutzt werde, und eine, mit der Zusteller und Kunden arbeiteten. Die Einfuehrung der fuenfstelligen Postleitzahlen sei hier kaum moeglich, da den Unternehmen nur die amtlichen Postdaten zur Verfuegung gestellt wuerden. Erst nach einigem "Nachhaken" habe sich der Postdienst bereitgefunden, der Techniker Krankenkasse inoffiziell eine Liste zur Verfuegung zu stellen, auf der beide vierstelligen Postleitzahlen verzeichnet seien. "Die deutsche Wirtschaft hat auf die Postleitzahlen-Umstellung nicht gewartet" Dieter Struve, Systemanalytiker beim Neckermann Versand in Frankfurt Die hoechsten Kosten fallen bei der Krankenkasse wie in den meisten anderen Unternehmen bei der Programmierung an. In zahlreichen Firmen sind mehrere tausend Einzelanwendungen umzustellen. Dabei handelt es sich vorwiegend um Cobol-Programme, in denen zum Beispiel die Deklarationen solcher Variablen anzupassen sind, die die Postleitzahl enthalten. Auch die Datenstrukturen, Redefines und Renames koennen betroffen sein. Auf die Umstellung dieser Programme haben sich verschiedene Softwarehaeuser spezialisiert, darunter auch die Maas High Tech Software GmbH in Stuttgart. Schwierigkeiten, so die Erfahrungen der Softwerker, entstehen vor allem dann, wenn die Programmlogik nicht bekannt ist. Das ist in solchen Unternehmen der Fall, in denen die Dokumentation komplexer Anwendungen ausser im Kopf weniger, zum Teil bereits ausgeschiedener, Mitarbeiter gar nicht oder nur unvollstaendig vorliegt. Interne funktionale Abhaengigkeiten von Variablen - zum Beispiel ueber Hilfsvariablen - werden in Programmen haeufig nicht gesehen. Zudem muessen in der Regel Aenderungen in der Programmlogik vorgenommen werden, da mit den Postleitzahlen in vielen Faellen geografisch relevante Aussagen - etwa die Unterscheidung Ost- (O) oder Westdeutschland (W) - verbunden wurden. All diese Aenderungen kosten Zeit und Geld, doch an beidem fehlt es vielen Unternehmen angesichts der wirtschaftlichen Situation. Bereits im August letzten Jahres hat die Neckermann Versand AG in Frankfurt mit der Bereinigung ihrer Postleitzahlen begonnen. Das Unternehmen steht unter besonderem Druck, da es seine rund 19 Millionen Adressen nicht erst zum 1. Juli 1993 umzustellen hat. Der neue Katalog soll die Kunden schon am 3. Juli dieses Jahres unter der neuen Anschrift erreichen. Da

der Andruck bereits sechs Wochen vorher beginnen muss, hat das Unternehmen den Wechsel schon weitgehend bewaeltigt. Obwohl Neckermann auf die Zusammenarbeit mit externen Partnern setzt, sind immerhin rund 20 Entwickler, Wartungsprogrammierer und Organisationsfachleute fuer knapp ein Jahr gebunden. Wie nahezu alle Grossunternehmen hat auch das Versandhaus einen Schattenbestand an Adressen aufgebaut, der zum 1. Juli die alten Anschriften komplett ersetzen soll. Da die Adressen schon immer auf dem postalisch neuesten Stand gehalten wurden, lag die Menge der manuell zu korrigierenden Daten bei unter einem Prozent. Gravierender wirkt sich auch hier bis heute die Korrektur der bestehenden Anwendungen aus. "Wir muessen die ganze Datenbank noch einmal laden, den Matchcode neu erstellen, Tabellen anlegen und etliche Programme aendern", beschreibt Postleitzahl-Experte Struve die Dimension des Projektes. Schwierigkeiten entstehen seinem Haus wie so vielen Grossunternehmen nicht zuletzt deshalb, weil mit der alten Postleitzahl ein Ordnungskriterium abhanden gekommen ist, das als geografische Orientierungsgroesse unverzichtbar schien. Waren die vierstelligen Zahlen noch gebietsbezogen organisiert, so genuegen die fuenstelligen Postleitzahlen in erster Linie funktionalen Kriterien. Orte, die bisher postalisch getrennt waren, werden zusammengelegt, andere Bezirke, die nach Postkriterien eine Einheit bildeten, werden neu aufgeteilt. Noch nicht einmal die ersten beiden Ziffern der alten und neuen Postleitzahlen sind in jedem Fall kongruent. Damit ist etwa eine Postleitzahl-bezogene Zuordnung von Vertriebsgebieten, von Tourenplanungen fuer Verkaufsfahrer und Techniker, von Umsatzstatistiken oder von Provisionsabrechnungen weitgehend ausgeschlossen. Dasselbe gilt fuer die Erhebung soziodemografischer Daten, die von den Behoerden ebenso wie von Wirtschaftsunternehmen benoetigt werden. Dazu zaehlen zum Beispiel Kriminalstatistiken, Informationen ueber die Alterstruktur in bestimmten Gebieten oder auch Bevoelkerungs- und Arbeitsstaetten- Daten. Ueber dieses Problem ist in der Vergangenheit viel gestritten worden. Die von der Post angestrebte Loesung liegt in der Verwendung eines von den Postleitzahlen unabhaengigen zusaetzlichen Kriteriums, des elfstelligen Kreisgemeinde-Schluessels. Die Behoerde liefert in ihren Dateien einen solchen Schluessel mit. Immer wieder ist jedoch zu hoeren, dass die Postkunden damit nichts anzufangen wissen. Entscheiden sie sich dann doch fuer seine Verwendung, so ist die Enttaeuschung gross: Der Schluessel ist nicht vollstaendig, er kann in der angelieferten Form nur begrenzt genutzt werden. Softwarehaeuser und Dienstleister wie etwa die Wiese & Partner GmbH in Neu-Isenburg oder das Rhein-Main-Rechenzentrum in Frankfurt profitieren von diesem Missstand. Ihre Dienstleistung besteht nicht nur in der Konvertierung der Postleitzahlen, sie fuegen in die Adressdateien ihrer Kunden zusaetzlich den Kreisgemeinde-Schluessel als geografisches Ordnungskriterium ein - und zwar einen vollstaendigen. "Ein nicht unerheblicher Teil der uns bekannten Softwarehaeuser ist unserioes" Wilhelm Huebner, Vorsitzender des Verbandes der Postbenutzer e.V. in Offenbach Waehrend Versicherungen, Banken, Behoerden und Industrieunternehmen die Postleitzahl-Umstellung generalstabsmaessig planen und die Kosten relativ genau kalkulieren, wissen viele Mittelstaendler heute noch immer nicht, was eigentlich auf sie zukommt. Wilhelm Huebner, Vorsitzender des Verbandes der Postbenutzer e.V. in Offenbach, sieht diese Betriebe als die Leidtragenden der Postleitzahl-Reform - denn sie fallen unserioesen Anbietern von Konvertierungs-Tools zum Opfer. "Ein nicht unerheblicher Teil der uns bekannten Softwarehaeuser ist unserioes", urteilt Huebner. Einige der Anbieter wuessten noch nicht einmal, wie nach Vorschrift der Post die neuen Adressen aufgebaut sein muessen, damit sie spaeter automatisch gelesen werden koennen. Welch hanebuechenen Unsinn manche Hersteller ihren Kunden verkaufen, macht Huebner an einem Beispiel deutlich: "Ein uns bekanntes grosses Softwarehaus erzaehlt seinen Kunden: ‚Mit der Umstellung koennt Ihr Euch bis 1994 Zeit lassen, es reicht aus, vor die vorhandene Postleitzahl eine fuehrende Null zu setzen. Wer das macht", so Huebner, "riskiert, dass alle seine Briefe in Sachsen- Anhalt und Thueringen ankommen. Das ist schon kriminell!" Der Stichtag 1. Juli 1993 wird von vielen Unternehmen offenkundig nicht ernstgenommen - eine Tatsache, die nach Ansicht Huebners fuer einen erheblichen volkswirtschaftlichen Schaden sorgen kann. Geruechte, nach denen die Post gar nicht in der Lage sei, die neuen Zahlen maschinell zu lesen und zu verarbeiten, haben dazu gefuehrt, dass sich viele

mittelgrosse Postkunden mit der Umstellung Zeit lassen. Die Aussage der Post ist jedoch eindeutig: "Wir werden vom 1. Juli an vorrangig Sendungen mit den neuen Postleitzahlen verarbeiten", betont Fraenzi Koske, Sprecherin der Generaldirektion Postdienst in Bonn. Briefe, die mit den alten Postleitzahlen versehen sind, werden voraussichtlich deutlich laenger unterwegs sein. Dass die volle Bandbreite der Moeglichkeiten, die das neue Postleitzahl-System biete, nicht von einem Tag auf den anderen genutzt werden kann, ist fuer Koske eine Selbstverstaendlichkeit. Dafuer muessten saemtliche Verteilmaschinen ad hoc umgeruestet werden - nach Aussage der Post-Sprecherin eine "Milliardeninvestition". Man werde bis zum Ende des Jahrzehnts die Logistik fuer den Briefdienst optimieren und entsprechende Maschinen einsetzen. Schon heute boeten allerdings die neuen Postleitzahlen klare Vorteile gegenueber den alten - zum Beispiel, indem eine guenstigere Fachbelegung in den vorhandenen Maschinenanlagen moeglich werde. Koske argumentiert routiniert. Die harsche Kritik der letzten Monate hat inzwischen nicht nur sie, sondern auch die meisten anderen Sprecher des Bonner Bundespost-Postdienstes abgehaertet. Vorwuerfe waren unter anderem wegen der stark verzoegerten Auslieferung von Test- und Leitdateien sowie deren fehlerhaftem Zustand aufgekommen. Doch auch die mangelhafte Informationspolitik der Post, die unzureichende Unterstuetzung ihrer Kunden bei der Umstellung und die hohen Preise fuer die unterschiedlichen Datentraeger - die nun einmal jedes Unternehmen benoetigt - waren Gegenstand der Kritik. Gravierend ist aus Sicht der Grossunternehmen jedoch vor allem ein Versaeumnis: die schlechte Kommunikation nach aussen. "Man haette frueher ankuendigen muessen, was mit den neuen Postleitzahlen eigentlich auf uns zukommt und welche Zeitplaene sinnvoll sind", moniert Laskawy von der Techniker Krankenkasse. "Auch die Bereitstellung der zur Umstellung notwendigen Leitdateien haette besser geplant sein koennen." Weil lange Zeit unklar war, wie die Dateien, die von der Post angekuendigt waren, organisiert sein wuerden, konnten die Verfahren fuer die Umstellung erst sehr spaet erarbeitet werden. "Die Post hatte uns nicht mitgeteilt, dass sie nicht nur die Feldlaenge der Postleitzahlen, sondern auch die der Orts- und Strassenangabe veraendern wuerde", aergert sich Laskawy. "Wir haben erst sehr spaet davon erfahren, naemlich im Oktober 1992 mit der Veroeffentlichung des Vorgehensmodell fuer die maschinelle Umstellung von Adressdateien auf die neuen fuenfstelligen Postleitzahlen." Dennoch weiss der Mitarbeiter der Techniker Krankenkasse, dass sein Unternehmen mitziehen muss, wenn es nicht dauerhaft Schaden nehmen will. Wilhelm Huebner, Sprecher der Postbenutzer, bringt auf den Punkt, warum es sinnlos ist, die neuen Zahlen zu boykottieren: "Wenn die Post auf die Nase faellt, nimmt nicht nur sie Schaden. Es sind in erster Linie die Kunden, die betroffen sind." Heinricht Vaska

Dies ist der Auszug aus der Webseite [web7.5]:

Am 1. Juli 1993 steht in vielen Unternehmen die DV- Welt kopf Die fünfstelligen Postleitzahlen bereiten DV-Chefs Kopfzerbrechen

COMPUTERWOCHE Nr. 50 vom 11.12.1992

MÜNCHEN (hv) - Der europäische Binnenmarkt wird nicht das einzige Großereignis bleiben, mit dem sich deutsche DV-Verantwortliche im nächsten Jahr auseinandersetzen müssen. Eine zusätzliche Herausforderung bedeutet die Einführung der neuen fünfstelligen Postleitzahlen zum 1. Juli 1993, die aufgrund der deutschen Einheit notwendig geworden ist. Daß die Umstellung vorhandener Datenbestände und Programme für viele Unternehmen kompliziert werden könnte, darauf deuten schon die nackten Zahlen hin: Die Menge der Postleitzahlen wird von gegenwärtig 5300 auf rund 34 000 steigen. Davon entfallen 14 000 auf Ortsbereiche und neuerdings 20 000 Nummern auf Postfächer und Großkunden. Ortsteile von Städten werden ab Mitte nächsten Jahres in mehrere Postleitzahlbezirke aufgeteilt. Maßstab ist dabei die Anzahl der für ein Gebiet vorgesehenen Briefträger. Diese Neuregelung kann im DV-Bereich vor allem Anwender mit großen Adreßbeständen hart

treffen. Sie stehen unter dem Druck, ihre Daten termingerecht zu aktualisieren, denn, obwohl die Bundespost zumindest theoretisch eine Übergangsfrist einräumt, wird sich ab 1. Juli die Beförderungsdauer von Briefen mit alter Postleitzahl voraussichtlich stark verzögern. Die Post bietet lediglich das neue Straßen- und Orts- beziehungsweise Postleitzahlverzeichnis auf Datenträgern an - für die Umstellung und den Abgleich der Daten müssen die Anwender selbst sorgen. Umstellung kann Millionenaufwand bedeuten Welchen Aufwand eine solche Konvertierung bedeuten kann, verdeutlicht das Beispiel des Quelle-Konzerns. Das Versandhaus veranschlagt rund neun Millionen Mark für die Neuaufbereitung ihres Adreßbestandes, der angeblich bei rund 27 Millionen Eintragungen liegen soll. Zwar bildet ein Datenvolumen dieser Größenordnung eher die Ausnahme, doch Quelle ist längst nicht mehr das einzige Unternehmen, das inzwischen eine vielköpfige Projektgruppe mit der Umstellung betraut hat. Trotz des spürbaren Engagements bei sehr großen vertriebsorientierten Unternehmen lassen Aussagen von Software- und Beratungsunternehmen darauf schließen, daß sich Anwender dem Thema bisher nur unzureichend widmen. Oft werden die Auswirkungen der neuen Postleitzahlen auf die hauseigene Datenverarbeitung verkannt, die Unternehmen beginnen erst spät - möglicherweise zu spät - mit der Korrektur ihrer Datenbestände. Vielen Usern scheint nicht klar zu sein, daß von der neuen Postleitzahlregelung nicht allein die Adreßbestände, sondern auch komplexe Anwendungen betroffen sind. So müssen etwa die jeweiligen Feldlängen für Orts- und -Straßennamen den geänderten Vorgaben des Postdienstes angepaßt werden. Außerdem sind die Datensätze um ein zweites Postleitzahl-Feld für Postfächer zu erweitern. Beeinträchtigt werden können auch Teile des Formular- und Druckwesens, denn Geschäftspapiere, Visitenkarten, Kataloge, Publikationen, Stempel, Frankierautomaten etc. müssen rechtzeitig mit den neuen Adressen versehen werden. Auch Organisationsprobleme dürften sich einstellen, nämlich dann, wenn die vielerorts übliche Zuordnung von Verkaufs- oder Zuständigkeitsgebieten nach Postleitzahlen beeinflußt wird. Wichtig ist, daß für eine automatisierte Konvertierung der Datenbestände ein fehlerfreier, einwandfrei zu identifizierender Adreßbestand vorliegt. Die Neuordnung der Postleitzahlen hat nämlich unter anderem zur Folge, daß dort, wo etwa Stadtteile in mehrere Bezirke zerfallen, Adressen sich nur dann zuordnen lassen, wenn zusätzlich der Straßenname ausgewertet wird. Schreibfehler oder Abkürzungen aufgrund zu kurzer Datenfelder, die im Vorfeld der Konvertierung nicht bereinigt wurden, verhindern das schnelle Auffinden des jeweiligen Pendants in der Postdatei und machen so eine langwierige manuelle Bearbeitung erforderlich. Ab Februar 1993 gibt es die Umstellungsdateien Die Post stellt ihre acht Umstellungsdateien, die unter anderem die neuen Postleitzahlen und ein aktuelles Straßenverzeichnis enthalten, erst ab Februar 1993 auf CD-ROMs, Bändern und Disketten zur Verfügung - zu spät, wie viele Softwarehäuser meinen. Denn erst mit diesem Datenbestand von 80 MB können die bis dato unbekannten Daten analysiert und die Programme abschließend getestet und getunt werden. Heute steht für die Programmentwicklung seit wenigen Wochen ein Testdatenbestand von 7 MB zur Verfügung, der bei der Post bezogen werden kann. Informationspolitik der Post kritisiert Anwender und Softwarehäuser kritisieren nicht nur die späte Auslieferung der Daten, sie ärgern sich auch über die Informationspolitik der Bundespost. Mitarbeiter der ersten und zweiten Führungsebene bei Postkunden sowie DV- und Organisationsleiter sollten bis spätestens November 1992 über die Veränderungen in Kenntnis gesetzt werden. Bis heute scheint dieser Plan jedoch nicht in die Tat umgesetzt worden zu sein. "Unsere Kunden wurden nicht angeschrieben", kritisierte zum Beispiel Ulrich von Welck, Geschäftsführer der Münchner Angewandte Computer Software GmbH (ACS) im Gespräch mit Pressevertretern. Sein Unternehmen gehört - wie auch die TS Organisationsberatung für Vertrieb und Marketing GmbH in Neumarkt-St. Veit oder die IBS Systemvertrieb GmbH in Garbsen - zu einer Reihe kleinerer Softwarehäuser, die Anwendern mit Beratung und PC- oder Mainframe-Produkten aus der Bredouille helfen wollen. Dadurch, daß diese Anbieter ihren Kunden bei der Planung und Durchführung von Adreßumstellungen zur Seite stehen, springen sie für die Post in die Bresche. Der Postdienst stellt nämlich zur Zeit nicht mehr als drei DV-Berater ab. Diese Mitarbeiter dürften mit der Betreuung einiger weniger Großkunden genug zu tun haben.

Dies ist der Auszug aus der Webseite [web7.6]:

Neue Postleitzahlen geben den DV-Spezialisten einiges zu tun

COMPUTERWOCHE Nr. 12 vom 19.03.1993 Seite 145-147

Zum 1. Juli 1993 fuehrt die Bundespost - Postdienste - ein neues, gesamtdeutsches Postleitzahlen(PLZ)-System ein. Der hohe Verbreitungsgrad der DV macht diese Massnahme - im Gegensatz zur Einfuehrung des alten Systems (1961) - volkswirtschaftlich relevant. Manche Unternehmen kalkulieren mit zweistelligen Millionenbetraegen. Reinhold Voelker befasst sich mit den DV- technischen Aspekten der Umstellung.*

Die Aenderungen in der PLZ-Systematik sind mittlerweile weitgehend bekannt:

Alle die Leitung der Post bestimmenden Informationselemente sind in einer einzigen, nun fuenfstelligen Zahl zusammengefasst. Zusaetzliche Angaben zum Zustellpostamt (zum Beispiel Muenchen 81 oder Post A-Dorf) entfallen ebenso wie die Kennzeichnungen W und O.

Fuer Postfach-Zustellung und Empfaenger von Massensendungen gelten eigene Leitzahlen. Etwa 25 Prozent des Postaufkommens koennen damit automatisch in spezielle Sortiergaenge eingeschleust werden.

Die ersten beiden Stellen der neuen PLZs repraesentieren insgesamt 83 Regionen. Sie erfuellen hauptsaechlich postlogistische Zwecke, decken sich also nicht mehr mit Bundesland- oder Landkreisgrenzen. Ferner kennt die PLZ kuenftig die fuehrende Null (Dresden: 01, Leipzig: 04 etc.).

Die restlichen drei Stellen verschluesseln je Postort gegebenenfalls mehrere Zustellbezirke, Postfachschraenke und Grossempfaenger.

In 209 Staedten wird es mehr als eine Zustell-PLZ geben; die Abgrenzung erfolgt ueber Strassen, zum Teil auch Hausnummern. Auf diese Orte entfallen etwa 50 Prozent der deutschen Adressen.

Versteckte Fussangeln warten auf den Anwender
Auf den ersten Blick erscheint die Umstellung DV-technisch als leichte Uebung. Ein Datenfeld ist von vier auf fuenf Stellen zu erweitern. Die neue PLZ wird durch Standardsoftware oder in Service-Rechenzentren ermittelt. Die Post stellt dazu Leitdateien bereit.

In der Praxis werden aber Schwierigkeiten nicht ausbleiben. Schon die acht Umstellungsdateien der Post signalisieren eine gewisse Komplexitaet. Das einzelne Unternehmen wird je nach Situation (Menge und Qualitaet der Adressen, Struktur von Programmen und Daten) mit zahlreichen, oft noch nicht erkannten Fussangeln konfrontiert.

Das beginnt bei den Kosten. Standardsoftware zur inhaltlichen PLZ-Umstellung wird zum Kauf und im Service angeboten. Die Preise dafuer liegen zum Teil deutlich im sechsstelligen Mark-Bereich. Hinzu kommt eigener Aufwand zum Versorgen der In- und Output-Schnittstellen. Automatisches Aendern setzt voraus, dass die Schreibweise der Altadressen mit der in den Umstellungsdateien uebereinstimmt. Veraltete Bezeichnungen, unterschiedliche Abkuerzungen, Schreibfehler etc. muessen vorher bereinigt werden. Teilweise wird das nur manuell moeglich sein.

Die nun seit ueber 30 Jahren gueltigen Leitzahlen bilden nicht selten die Basis fuer organisatorische Strukturen. Beispiele dafuer sind Vertriebsgebiete, Betreuungszuordnungen und Tarifzonen. Elemente aus der Postleitzahl (zum Beispiel die erste Stelle) werden bisher in Plausibilitaetspruefungen, statistischen Aggregationen (Summen, Zeitreihen), Ableitungen und Zuordnungen benutzt.

Diese Strukturen sind an die Interpretierbarkeit der alten PLZ geknuepft (zum Beispiel war von der Zahlenlaenge auf die Groesse des bezeichneten Ortes zu schliessen) und lassen sich in das neue System nicht uebernehmen. Dezentrale Adressenbestaende zum Beispiel in PCs erfordern spezielle Konvertierungen oder manuelle Umstellung. Betroffen sind in Datenbanken und auch in Standardsoftware gespeicherte Adressen fuer Briefaktionen.

Auch die herkoemmliche Bueroorganisation ist hier gefordert: Formulare aller Art, Visitenkarten, Briefboegen, Stempel und Werbeabonnements muessen geaendert werden. Der Aussendienst muss den Kunden beispielsweise in 80654 Muenchen statt in Muenchen 70 oder Muenchen-Sendling aufsuchen.

Ein Testfall fuer die Software-Architektur
Selten zeigt sich die Bedeutung von klaren Systemstrukturen so deutlich wie jetzt. Der Aufwand fuer die PLZ-Umstellung ist ein Gradmesser fuer die Qualitaet der eingesetzten Software-Architektur.

Die PLZ ist kuenftig nicht mehr allein vom Zustellort bestimmt. Sie bezeichnet vielmehr meist eine detailliertere postalische Position als bisher und kann daher sogar vom Zustellwunsch des Empfaengers (wenn dieser zum Beispiel mehrere Postfaecher hat) abhaengen.

Nach dem Prinzip "One fact, one place" ist jedem Datenelement, also auch den Bestandteilen von Adressen, ein eindeutiger Ort im Informationsmodell zuzuweisen. Es darf logisch nur eine zentrale Adressen-Datenbasis geben, deren Bestaende strukturell redundanzfrei sein muessen. Adressen von Kunden, Mitarbeitern, Lieferanten, Versandzielen, Organisationseinheiten etc. finden dort ihren Platz.

"Partner" und "Adresse" sind getrennte Informationsobjekte (IO). Eine neue PLZ veraendert nicht die Einzelobjekte von "Partner": Herr Schmitt bleibt derselbe wie zuvor. Daraus ergibt sich, dass die PLZ Eigenschaft eines anderen Objekts sein muss: der "Adresse". Zwischen beiden Objekten besteht die Beziehung ",Partner hat Wohnsitz an ,Adresse". Durch diese Trennung lassen sich mehrere Adressen pro Partner fuehren.

Empfehlungen fuer die Planung
Die Datentechnik kennt unterschiedliche Mechanismen zur Herstellung von Beziehungen. Vor allem bei relationalem Database Management System (DBMS) sind Beziehungstabellen zweckmaessig. Sie sind fuer mehrere logische Beziehungen nutzbar, koennen unterschiedliche IOs verbinden, Attribute aufnehmen und belasten den Datenhaushalt bei optionalen Beziehungen nicht (vgl. die Abbildung). So kann etwa die Zustelladresse leicht vom Postfach unterschieden werden. Ueber eine einheitliche Struktur lassen sich alle Adressierungsanforderungen erfuellen und datentechnisch definieren.

Die Umstellung ist eine Chance, noch in Gruppen gefuehrte Datenfelder (zum Beispiel PLZ und Ort, Strasse oder Postfach, akademischer Titel und Anrede) elementar zu speichern. Ein zentrales Modul bereitet versandfertige Adressen auf.

Die objektorientierten Techniken gelten haeufig als unausgereift und praxisfern, ihre Vertreter gar als Scharlatane. Die PLZ- Umstellung mit diesen Methoden (kurz: OO) sollte im Prinzip nur in der Objektklasse "Adresse" eine Aenderung bewirken. Ob das gelingen kann, ist aber noch nicht beweisbar, da die Techniken, wie OO- Pioniere bestaetigen, meist noch nicht unternehmensweit integriert sind.

Fuer die PLZ-Umstellung sind die folgenden Kriterien Erfolgsfaktoren. Zum Grossteil sind sie auch Zielgroessen im Systems Engineering (SE). Kommt die SE-Realitaet des Unternehmens dieser Vision schon nahe, so wird der Aufwand fuer die Umstellung nicht allzu gross sein. In anderen Faellen kann der Wechsel als Chance begriffen werden, das SE zu verbessern.

- Die fachliche Zustaendigkeit fuer die Adressen und ihre Pflege ist klar geregelt.
- Adressen werden in einem zentralen Bestand gefuehrt. Die Umstellung betrifft nur eine Speichertechnologie.
- Dezentrale Bestaende sind automatisch abgeleitete Kopien. Als Folge der zentralen Aenderung werden sie automatisch aktualisiert, beduerfen also keiner eigenen Umstellung.
- Die Daten (hier: PLZ, Ort etc.) sind in Include-Strukturen definiert. Alle Operationen beziehen sich ausschliesslich auf die dort festgelegten Feldnamen. Die Programme reagieren auf geaenderte Feldlaengen automatisch und richtig; Eingriffe in die Programme selbst sind damit weitgehend ueberfluessig.
- Programme, die die PLZ nicht verwenden, sind von der Umstellung nicht betroffen. Bei Programmverbunden mit gemeinsamen, meist im Maximalformat ausgelegten Datenstrukturen genuegt die Kompilierung.
- Module sind extern ladbar. So entfallen aufwendige Bindeprozeduren, und das Uebergabeverfahren ist einfach.
- Ein Dictionary zeigt die Zusammenhaenge von Daten, Programmen, Masken, Listen etc.
- Vor der Umstellung sollte man unbedingt die alten Adressen aktualisieren. Nur korrekte Altadressen erlauben es, einen Grossteil der Umstellung automatisch durchzufuehren.
- Das Umstellungsteam muss fruehzeitig installiert und das Projekt solide geplant werden, um den Umfang der Massnahme rechtzeitig zu erkennen und zu definieren. Es ist bereits hoechste Zeit dafuer.
- Die Umstellungsstrategie ist sorgfaeltig auszuwaehlen: Welche Umstellungssoftware empfiehlt sich? (Eigenentwicklung ist kaum die richtige Loesung). Sind Schattenbestaende erforderlich? Moechte man kuenftig Adressaenderungen qualifiziert pruefen? Wie?
- Bei Bedarf sind integrierende Massnahmen erforderlich: Feldlaengen erweitern, Kleinschreibung und Umlaute einfuehren, zentralen Bestand schaffen. Als Basis fuer regionale Klassifizierungen kann der Gemeindeschluessel eingefuehrt werden.
- Das organisatorische Umfeld muss rechtzeitig einbezogen werden.

Sehr starke Belastungen

Die PLZ-Umstellung wird viele Unternehmen sehr belasten. Sie bindet Entwicklungsressourcen, die fuer geschaeftspolitisch wichtigere Aufgaben dringend noetig waeren. Wenn sie ausgerechnet in die Strukturen der aeltesten, vielleicht noch wenig standardisierten Bestaende - und damit in zahlreiche Programme - eingreift, so sollte das nur nach methodisch gesicherten Konzepten geschehen. Die Aenderung erschoepft sich nicht im Einsatz der Umstellungsprogramme. Vielmehr muss den langfristigen Erfolg der SW-Architekt sichern; er integriert auch in der Wartung.

*Reinhold Voelker ist Management-Berater bei der Ploenzke Informatik AG in der Geschaeftsstelle Muenchen/Finanzinstitute.

Abb: Das Informationsobjekt "Adresse" und sein Umfeld: Konzeptionelles Datenmodell und Physik muessen nicht immer identisch sein. Quelle: Ploenzke