„CASE“ findet im Kopf statt

- Der Einsatz von Konzepten und Werkzeugen -

Thorsten Spitta
Universität Bielefeld
thspitta@wiwi.uni-bielefeld.de

Zusammenfassung. Der Beitrag betrachtet Software-Entwicklung und -pflege in der Praxis. Er zeigt, wie auch in klassischen Umgebungen nach Software-Engineering-Prinzpien gearbeitet werden kann. Dazu werden sechs Thesen zu CASE aus der Sicht eines Anwenders aufgestellt, der Software auch *warten* muß. Die Thesen dienen als Maßstäbe zur Beurteilung von Werkzeugen. Sie werden anhand von drei Fallstudien überprüft. Die Fallstudien referieren werkzeuggestützte Softwarekonzepte, die sich seit mindestens fünf Jahren im Einsatz befinden. Die Beispiele zeigen, wie man mit zum Teil einfachen Mitteln ein Verständnis aller Entwickler für die Prinzipien des Software Engineering schaffen und ihre praktische Anwendung erreichen kann. Selbst die Unüberschaubarkeit von Altsystemen muß nicht schicksalhaft hingenommen werden, sondern ist durch Konzepte und Werkzeuge änderbar. Wenn der Nutzen des Software Engineering verstanden wird, dann ist die geistige Basis geschaffen, die eine wirtschaftliche Anwendung anspruchsvollerer Techniken wie etwa der Objektorientierung voraussetzt.

1 Positionsbestimmung

Die Rezession 1993/94 in Deutschland hat erstmals in der Geschichte auch die Informationstechnik selbst zum Gegenstand einschneidender Rationalisierungen gemacht. Softwareentwicklung und -wartung war plötzlich ein Geschäftsprozeß wie viele andere auch, den man „optimieren“, mit anderen Worten *rationalisieren* kann. Dies hatte die Disziplin *Software Engineering* schon 25 Jahre lang mit mäßigem Erfolg behauptet. Waren etwa zu wenig *CASE-Werkzeuge* eingesetzt worden oder hatte man am Bedarf vorbeientwickelt?

Die 1992 ausgewertete Studie von Welz und Ortmann, in der 48 mehrjährige Softwareprojekte untersucht wurden [WeOr92], liefert reichlich Material für Erklärungen:

Die Kernbotschaft des Software Engineering, daß Softwareentwicklung ein zu planender, ständig zu kontrollierender und zu validierender Prozeß sei, wird nicht verstanden oder nicht ernst genommen. Dieser Prozeß läuft nicht nach einfachen Phasenschemata ab und kann nicht durch disziplinierende CASE-Werkzeuge Erfolg haben. Vielmehr sollte die Fähigkeit von Entwicklern und Managern verbessert werden, grundlegende Konzepte in ständig wechselnden Situationen anzuwenden.

Der Autor hat in einer großen und einer mittelständischen Firma Software Engineering eingeführt (Schering AG und Vatter GmbH -Strümpfe; 500 MioDM Umsatz-) und dabei Softwarewerkzeuge und Anwendungen entwickelt oder beschafft [Spi83; Spi89, Kap.11]. Die Fehlschläge des Software Engineering in der Praxis beruhen nicht auf mangelnder Investitionsbereitschaft oder falschen Konzepten, sondern darauf, daß der nachweisbar größte Produktivitätsfaktor, der *Kopf* des Entwicklers [Boe87], nicht sinnvoll eingesetzt wird. Konkreter heißt dieses: Es werden elementare Konzepte des Software Engineering vernachlässigt, hierdurch die Entwickler mit manuellen Routinetätigkeiten überfordert und ihre Kreativität für die Anwendungslösung blockiert. Dies belegt auch die empirische Studie von Hesse und Frese zur Arbeitssituation von Softwareentwicklern [HeFr94].

Leider gibt es zu wenige empirische Untersuchungen zum Einsatz von CASE-Werkzeugen. An ihre Stelle treten Fallstudien, die vom Einsatz produktiver Systeme berichten. Solche Fallstudien erscheinen nur dann angemessen, wenn man über die Symbiose zwischen Konzepten und Werkzeugen spricht. Die hier referierten Fälle sind so ausgewählt, daß auf ein relativ ein-

faches ein etwas komplexeres und danach ein anspruchsvolles werkzeuggestütztes Konzept folgt:

1. Mit Bezeichnern, etwa von Attributen oder Variablen, fördert oder verhindert man eine unternehmensweite Sprachkultur. Beispiel 1 handelt von *Sprache als Grundelement von Software*.
2. *Modularität und Wiederverwendbarkeit* müssen praktisch beherrschbar sein. Beispiel 2 zeigt, wie man dies selbst mit Altsoftware auf einfache Weise vermitteln kann.
3. Nutzeffekte durch *Standardisierung* werden immer wieder versprochen. Beispiel 3 zeigt, wie man dies im Bewußtsein von Entwicklern „implementieren“ kann, und zwar lebensfähig über die technologischen Sprünge von nunmehr 10 Jahren.

Der Bericht knüpft bewußt an elementaren Prinzipien an, weil gerade diese häufig so abstrakt bleiben, daß sie keine praktische Wirkung haben. Die Beispiele zeigen vor allem, wie man zwei Grundanliegen des Software Engineering umsetzt, die für alle weiteren Fortschritte unabdingbar sind:

- *Übergeordnete Strukturen* sichtbar und damit für die Organisation beherrschbar machen,
- *Wiederverwendung* auf allen Ebenen fördern. Wiederverwendet werden sollen Ideen, Konzepte, Bausteine und organisatorische Prozesse.

Erst wenn die elementaren Konzepte von *allen* Entwicklern praktiziert werden, ist der Nährboden für anspruchsvollere Techniken wie etwa der Objektorientierung bereitet. Den jüngsten empirischen Beweis, daß es immer noch an Elementarem mangelt, lieferte am 1.1.1996 die Deutsche Telekom: Es ist nicht das Problem, daß ein vermutlich unter hohem Zeitdruck arbeitender Programmierer einen geradezu entsetzlich banalen Programmierfehler beging, indem sein Programm am Neujahrstag die Gebühren nach dem Tarif eines Werktages abrechnete. Der Skandal ist, daß diese Software, die Millionen Nutzer hat, offensichtlich für ihren *ersten* Einsatztag nicht getestet worden war.

Damit die Fallstudien nicht zu kasuistisch im Raum stehen, werden sechs Thesen zu CASE formuliert, die für den Leser als Maßstäbe bei der Beurteilung von CASE-Werkzeugen dienen sollen und in den Fallstudien auch als solche benutzt werden.

2 Sechs Thesen zu „CASE“

CASE für *computer aided software engineering* ist zwar eine praktische Abkürzung für *Werkzeugbenutzung*, also eine Selbstverständlichkeit, wird aber zum Täuschwort [Gra93], wenn damit höherwertige Qualität und Produktivität suggeriert werden soll. Praktisches Software Engineering muß immer werkzeugunterstützt ablaufen. Wäre es anders, würden die härtesten Rationalisierer ihre eigenen Prinzipien mißachten. Die Anführungszeichen im Titel sind so zu verstehen, daß sich der Autor von der Modesemantik der Abkürzung CASE abheben möchte, sie aber dennoch benutzt, weil es praktisch und inzwischen auch üblich ist.

Die folgenden Thesen zu CASE sind als Anforderungen an Werkzeuge und ihren Einsatz gedacht. Ihre Beachtung fördert die in Abschnitt 1 angesprochene Bewußtseinsbildung und eliminiert Modewerkzeuge. Um die Thesen besser referenzierbar zu machen, werden ihnen Eigenschaften zugesprochen, die Werkzeuge und ihr Einsatzkonzept aufweisen müssen. Ausführlichere Darstellungen zu Werkzeugen mit anderen Blickwinkeln finden sich bei Denert und Wallmüller [Den91, Kap. 18; Wall90, Abschn. 3.5]. Unter „CASE“ wird hier also *Werkzeugeinsatz auf Basis vorher hinterfragter Konzepte* verstanden.

Tab. 1: Anforderungen an CASE

Nr.	Eigenschaft	These
1	*kommunikativ*	Zusammenarbeit fördern, nicht Virtuosentum
2	*pragmatisch*	Perfektionismen vermeiden
3	*allgemeingültig*	Schwerpunkte setzen und durchsetzen
4	*ganzheitlich*	Die Wartung unterstützen, nicht (nur) die Entwicklung
5	*akzeptierbar*	Auf Verständlichkeit achten und Wirtschaftlichkeit belegen
6	*empirisch*	Maße zur Objektivierung liefern

(1) Zusammenarbeit fördern, nicht Virtuosentum. Bei der Einführung von Software Engineering wird oft eine elitäre Strategie verfolgt. Dies ist nicht sinnvoll. Vielmehr müssen junge und evtl. besser ausgebildete Mitarbeiter schon in den Pilotprojekten gemeinsam mit älteren und eher nachgeschulten Kräften eingesetzt werden. Daneben müssen Werkzeuge nicht nur die Arbeit innerhalb von Teams unterstützen, sondern die technologische Integration und die Kommunikation *zwischen* Teams fördern. Dies wird mit allein workstation-basierten Werkzeugen nicht möglich sein, sondern nur mit einer ergänzenden, serverbasierten Entwicklungs-Datenbank. Nur wenn Entwickler sich mit den Ideen anderer auseinandersetzen müssen, kommt eine Bewußtseinsveränderung in Gang.

(2) Perfektionismen vermeiden. Viele Methoden und Werkzeuge werden auf Basis kleiner Beispiele entwickelt, manchmal ohne Erfahrungen der Entwickler mit realen Anwendungen. Vor allem Fachentwurf und Spezifikation können so schnell aufwendig und unübersichtlich werden. Die Paradigmen *wissenschaftliche Genauigkeit* und *ingenieurmäßige Entwicklung* stehen sich zum Teil konträr gegenüber. Im Bewußtsein dieses Widerspruches sollte man Methoden auf ihre großtechnische Einsetzbarkeit prüfen und im Zweifel den effizienteren aber möglicherweise auch etwas unschärferen den Vorzug geben.

(3) Schwerpunkte setzen und durchsetzen. Die Methoden- und Werkzeuglandschaft ist selbst für Spezialisten kaum noch durchschaubar. Deshalb sollten Schwerpunkte gesetzt und danach ausgewählt werden, welchen Beitrag man zur Qualitätserhöhung und zur Kontrolle der Altsoftware erhält. Zu viele oder zu komplexe Werkzeuge sind kontraproduktiv [HeFr94]. Hat man jedoch einmal entschieden, hat die Verbindlichkeit für *alle* Entwickler oberste Priorität. Ausnahmeregelungen für eilige Projekte signalisieren, daß das Management nur halbherzig zu den getroffenen Entscheidungen steht.

(4) Die Wartung fördern, nicht (nur) die Entwicklung. Dieser Aspekt wird von den in der Softwaretechnik besonders aktiven Softwarehäusern wenig behandelt [vgl. z.B. Den91]. Es ist unstrittig, daß der größte Anteil der Softwarekosten auf die *Evolutionsphase* fällt („Wartung“). Das Denken von Entwicklern beim Anwender ist durch die Wartungserfahrung geprägt. Daher muß jeder Ergebnistyp, den ein Werkzeug hervorbringt, daraufhin überprüft werden, ob er nur der Entwicklung dient oder dauerhaft gewartet werden kann. Die Prüffrage ist nicht, *wie entsteht ein Ergebnis?*, sondern *wie wird eine Änderung abgewickelt* bzw. *wie finde ich ein Objekt und seine Beziehungen zu anderen Objekten?* Nur über die Betrachtung der Wartung werden einige Werkzeuge wie Projektbibliothek und Data Dictionary begründbar, die sich aus der Entwicklung nicht rechtfertigen lassen und auch sehr unbeliebt sind [HeFr94].

(5) Auf Verständlichkeit achten und Wirtschaftlichkeit belegen. Die Studie von Hesse et. al. hat deutlich gezeigt, welch große Bedeutung die Akzeptanz bei der Benutzung von Werkzeugen hat. Die Akzeptanz wird maßgeblich bestimmt von den Eigenschaften *leichte Erlern- und Bedienbarkeit*, in nur geringem Ausmaß von der *Funktionalität*. Akzeptanz heißt nicht, es

100% aller Mitarbeiter recht machen zu wollen. Gerade auch aus Akzeptanzgründen muß die *Wirtschaftlichkeit* von Werkzeugen nachgewiesen werden.

(6) Maße zur Objektivierung. Viele technologische Diskussionen sind von persönlichen Einschätzungen bis hin zu Emotionen geprägt. Zu dem vielschichtigen Thema Maße und Metriken sei an dieser Stelle lediglich festgehalten:

- Zeitkontierungen in Entwicklung und Wartung sind eine Grundbedingung für jede Wirtschaftlichkeitsbetrachtung [vgl. Spi89, Kap.12].
- Werkzeuge sollten Prozeß- und Bestandsdaten liefern.
- Werkzeuge müssen entweder nachprüfbare Referenzen aus praktischem Einsatz vorweisen oder in Pilotprojekten überprüft werden. Ein Pilotprojekt ist ein Experiment, das auch scheitern kann. Bei a priori getroffenen Managemententscheidungen bekommen Pilotprojekte eine Alibifunktion.

3 Konzepte, Werkzeuge und ihr Einsatz

Die folgenden drei Beispiele zeigen die Vermittlung von Konzepten und die Anwendung der Thesen. Sie stellen Lösungen zu folgenden Themen vor:

1. **Wiederverwendbare Attribute.** Attribute sind besonders wichtige Elementarbausteine von Informationssystemen. Sie entstehen in Datenmodellen, gehen ein in Spalten von Datenbanktabellen und werden benutzt in Variablen von Programmen. Hier überwiegt der Methodenaspekt, während das Werkzeug fast trivial ist. Die Wirkung ist gemessen am geringen Aufwand sehr groß.
2. **Aktive Referenzen für Altsoftware.** Das Konzept Modularität kann nicht nur für Neuentwicklungen gelten. Mangels eines käuflichen Werkzeuges wurde ein Source-Analysator entwickelt, der jederzeit aktuell die externen Referenzen aller Programme und damit auch die Wiederverwendung neu entwickelter Unterprogramme zeigt. Hier wurden mit wenig Aufwand undurchschaubare Strukturen offengelegt und der Wert von Modularität gezeigt.
3. **Eine implementierte Musterarchitektur.** Das Konzept einer wiederverwendbaren Software-Architektur wird durch Werkzeuge umgesetzt, die „evolutionäres Prototyping“ erlauben. Es wurde als verbindliche Basis für alle neuentwickelten Dialogsysteme benutzt, die in drei Firmen 200 Anwendungssysteme mit über 30.000 Modulen umfassen.

4 Wiederverwendbare Attribute

Kaum jemand zweifelt heute daran, daß Datenmodelle eine besonders wichtige Methodik sind, um Informationssysteme zu entwerfen. Seit es relationale Datenbanksysteme gibt, kann man ein Relationenmodell sogar ohne Strukturbruch als Datenbankschema implementieren. Eine solche durchgängige Konstruktion über drei Entwicklungsphasen zeigt Abbild 1:

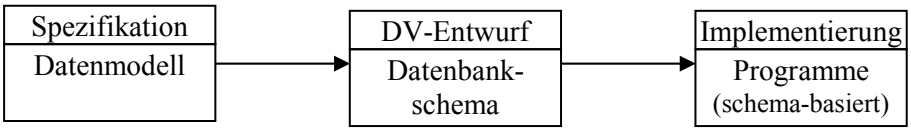

Abb. 1: Homogene Konstruktion vom Datenmodell bis zur Implementierung

Es gibt jedoch kein Konzept, wie man mit den elementarsten Bausteinen einer Datenbasis, den Attributen bzw. Datenelementen, sparsam umgeht und dies in einer Organisation durchsetzt. Dies geschieht durch Wiederverwendung *innerhalb* des Datenmodells und *zwischen* den Ergebnistypen der Phasen. Die methodische Vorgehensweise zur Erstellung von Datenmodellen wird seit langem gelehrt [Vett82, Den91] und auch in großen Unternehmen praktiziert [Han-

f91]. Ziel sind *Unternehmens-Datenmodelle* (UDM), die stufenweise aus Projekt-Datenmodellen entstehen.

Ein UDM hat mindestens 300, sehr schnell bis zu 2.000 Entitätstypen. Jeder Entitätstyp hat grob geschätzt 10 *Attribute*. Dann hat ein UDM bis zu 20.000 Attribute, wenn die Attribute disjunkt sind. Da aus Attributen bei Verwendung verschiedener Programmiersprachen mehrere *Datenelemente* und aus diesen wiederum repräsentationsabhängige *Felder* entstehen können, weist eine betriebliche Datenbasis schnell 100.000 und mehr Elementareinheiten auf, die sich selbst mit einem Data Dictionary nicht mehr überschauen lassen. Der Ansatzpunkt zur Komplexitätsreduktion ist die *Wiederverwendung von Attributen* als den kleinsten bedeutungstragenden Einheiten eines UDM. Das Relationenmodell sagt hierüber nichts aus. Ein Beispiel:

```
LAGER     (Lager-Nr, Bezeichnung, Anlegedatum)
BESTAND   (Lager-Nr, Artikel-Nr, Gesamtmenge, Änderungsdatum,
          Änderungszeit)
BEWEGUNG  (Lager-Nr, Artikel-Nr, Datum, Uhrzeit, Menge)
```

`Gesamtmenge` ist vom gleichen Typ wie `Menge`, entsteht hier sogar durch Addition aus Attributen der Art `Menge`. Wir sollten also `Menge` wiederverwenden, etwa in einer Integritätsbedingung:

```
∀ BEWEGUNG.Menge •
    BESTAND.Menge:= BESTAND.Menge + BEWEGUNG.Menge.
```

Die Qualifizierung von Attributen durch den Entitätstyp ist syntaktisch eindeutiger als eine Verbalqualifizierung im Namen. Ziel ist es, möglichst wenige, unternehmensweit überschaubare Attribute zu haben, die im Kontext verschiedener Entitätstypen wiederverwendet werden. Die Wiederverwendbarkeit bedingt, daß sich der Wertebereich nicht ändert.

Seit der Entwicklung von PASCAL (1971) benutzt man *Typen*, um Ordnung in semantische Sachverhalte zu bringen. Es werden drei Arten von *Attributtypen* unterschieden:

- ***Elementartypen*** wie REAL, INT, CHAR, die in jeder Programmiersprache vorkommen.
- ***Basistypen*** als semantische Typen über den Elementartypen wie etwa Faktor, Nummer, Anzahl.
- ***Rollen*** als Spezialisierungen von Basistypen. Sie teilen sich noch in *allgemeine* (z.B. Änderung, Korrektur, Abgang) und *anwendungsspezifische Rollen* (z.B. Charge, Variante, Alternative).

Es wird also ein Konzept aus der Welt der Programmiersprachen in die Welt der Datenmodelle übertragen, in der Attributtypen unbekannt sind. Dies ist ein Konzept der natürlichsprachlichen Namenbildung mit wenigen, allerdings wichtigen Restriktionen. Sie sind enthalten in den folgenden neun **Regeln zur Namenbildung**, die durch einfache online-Hilfen unterstützt wurden, als noch kein Data Dictionary zur Verfügung stand:

1. Namen *sprechend* oder *einheitlich abgekürzt*.
2. *Substantive*, wo möglich.
3. Es gilt ein *verbindlicher Katalog von Basistypen* mit festen Namen oder Abkürzungen.
4. Der *Basistyp ist als Suffix* zwingender Namenbestandteil. Paßt kein vorhandener Typ, muß der Typkatalog ergänzt werden.
5. Die *problemspezifischen Namenbestandteile* stehen in zunehmender Spezialisierung vorne.
6. Der Relationenname ist in der Regel *nicht* Bestandteil des Namens beschreibender Attribute, es sei denn, es wird ein spezialisierter Wertebereich begründet.
7. *Primärschlüssel* müssen datenmodell- bzw. datenbankweit eindeutig sein.

8. Für *zusammengesetzte Primärschlüssel* müssen *Sekundärschlüssel* vergeben werden, wenn sie als Fremdschlüssel benutzt werden.
9. *Namen* werden *durchgängig* vom Datenmodell über das Datenbankschema bis zu den Variablen der Programme benutzt, also *ohne ALIAS-Namen*.

Die folgenden Beispiele dürften auch ohne Erklärung der Abkürzungen verständlich sein. Die Namen sind weitgehend natürlichsprachlich, optisch gruppiert und damit lesbar und von akzeptabler Länge. Sie bilden eine problemspezifische Sortierfolge, was bei einer Präfix-Regel für die Typen nicht gegeben wäre.

Tab 2: Beispiele für Attributnamen mit Standard-Abkürzungen und Endungen für Basistypen

`Artikel-Nr`	`Lager-Abgangs-Mng`	`Bestands-Auslauf-Knz`
`Kunden-Nr`	`Auftrags-Storno-Anz`	`Einzelfaktura-Knz`
`Preis-Grp`	`Arbeitsgang-Bes`	`Fertigungs-Zust`
`Kd-eigene-Auftr-Nr`	`Arbeitsgang-Kurz-Bez`	`Gueltig-bis-Dat`
`Losgroessen-Mng`	`Gebinde-Kap`	`Fertigungs-Schr`

Die Basistypen sind in Tabelle 3 zusammengefaßt, wobei die Standard-Abkürzungen unterstrichen sind. Durch Analyse realer Dateien wurden 31 Basistypen gefunden. In Klammern stehen zulässige Synonyme:

Tab. 3: Zulässige Basistypen, gruppiert in *Typklassen*

Typklasse	**Basistypen** [... = offene Liste]
Identifikatoren	Key, Nummer, User-ID, Bezeichnung (Name), Beschreibung (Text)
Klassifikatoren	Code, Kennzeichen, Gruppe (Klasse, Art, Typ)
Zustände	Status (Zustand), Stufe (Schritt), Flag (?) [=bool], Steuerwert
Zeiten	Datum (Termin), Zeitraum (Dauer), Uhrzeit, ...
Mengen, Operatoren	Anzahl, Menge, Faktor, ...
Werte	Preis, Wert, Kosten, ...
Maße	Einheit, Gewicht, Volumen, Kapazität, ...

Bei Kenntnis der Basistypen bleiben im Beispiel nur noch zwei erklärungsbedürftige problemspezifische Standard-Abkürzungen: Kunde und Auftrag.

Das Konzept wurde im Zuge einer umfassenden Neuentwicklung in der Vatter GmbH eingesetzt, wobei der Nutzen der Wiederverwendung empirisch nachgewiesen werden konnte (detaillierter Bericht in [Spi96]). Die dezentral arbeitenden Entwickler brauchten nur eine sehr einfache Werkzeugunterstützung für ihre Namenvorschläge:

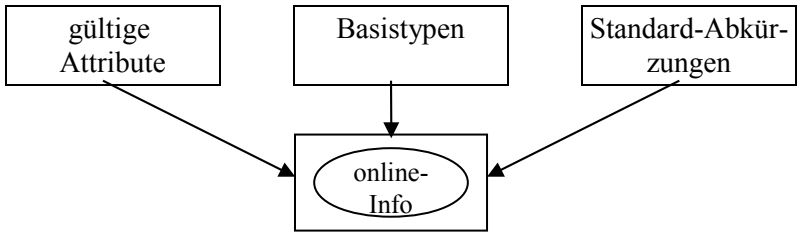

Abb. 2: Hilfe für den Entwickler bei der Namenbildung

Da die Namen frühzeitig in den Projekt-Datenmodellen entstehen, bleibt für die Datenadministration genügend Zeit, sie in das UDM und das zentrale Datenbankschema zu integrieren bzw. unerwünschte Synonyme zu entdecken. Die bekannte „Flaschenhals-Rolle“ der Datenadministration wird so vermieden. Homonyme sind aufgrund der semantisch deutlichen Namengebung nicht zu erwarten (und auch nicht vorgekommen). Zur Durchsetzung eines solchen Konzeptes ist ein Data Dictionary oder wenigstens ein zentraler Datenkatalog hilfreich.

Tab. 4: Mehr als sechsmal verwendete Attribute (Ausschnitt)

Attribut	**1992**	**1995**
Aend-Dat	56	87
Aend-User	40	82
Aend-UZeit	49	82
Apl-Pos-Key	21	0
Apl-Pos-Nr	0	31
Arb-Gang-Kurz-Bez	7	12
Artikel-Bez	7	13
Artik-Var-Key	3	14
Ausf-Werk-Key	6	10
Bestell-Mng	9	20
Buchungs-Dat	4	9
Buchungs-Pos-Nr	5	14
Einheit	22	35
Farb-Code	16	14
Gebinde-Art	14	35
Groessen-Code	16	14

Die Zahl der Datenelemente der Altanwendungen hatte bei nur einer Programmiersprache (COBOL) ca. 14.000 betragen. Die Neuentwicklung nach diesem Konzept ergab nur noch 807 Attribute (= Datenelemente), die zwischen 87 und einmal wiederverwendet wurden (s. Tab. 4 `Aend-Dat`). Der Ausschnitt aus dem Datenkatalog in Tabelle 4 zeigt den Grad der Wiederverwendung der 1988 begonnenen Neuentwicklungen bis Mitte 1995 (Apl= Arbeitsplan, Artik= Artikel, Ausf= Ausfertigung).

Durch welche Maßnahmen und Wirkungen entspricht das Konzept und das sehr einfache Werkzeug unseren Thesen?

Tab. 5: Umsetzung der Thesen in Beispiel 1

These Werkzeug ist ..	Umsetzung durch ..
kommunikativ	.. ansetzen an der elementarsten Kommunikationsform jedes Entwicklers, der Sprache. Attribute sind wie Objekttypen *Begriffe* und als solche Ausdruck der Sprachkultur einer Organisation.
pragmatisch	.. relativ große Freiheitsgrade bei der Namenbildung mit zulassen von Synonymen.
allgemeingültig	.. zwingende Einbindung der Datenadministration, die alleine Datenbank-Views generieren kann.
ganzheitlich	.. Erleichterung der Wartbarkeit der Programme. Die Selbstdokumentationskraft von Programmen hängt entscheidend von den Variablennamen ab.
akzeptierbar	.. Werkzeugunterstützung und Einbeziehung der Entwickler in die Weiterentwicklung. Es gab neben den Basistypen nur ca. 50 Standard-Abkürzungen.
empirisch	.. Auswertung und Bekanntgabe der Art der Wiederverwendung.

Die Namengebung spielt eine erhebliche Rolle für das technologische Bewußtsein der Entwickler. Software ist ein Sprachprodukt und prägt die Begriffskultur einer Organisation nachhaltig [Ort94]. Wird damit nachlässig umgegangen, signalisiert dies Geringschätzung der Unternehmensressource Software. Die Einbeziehung eines Typkonzeptes in die Variablennamen typloser Sprachen eröffnet vielfältige Möglichkeiten, die Daten besser beherrschbar zu ma-

chen, etwa allgemein implementierte Integritätsbedingungen oder die hier dargestellte Wiederverwendung.

5 Aktive Referenzen für Altprogramme

Viele Versuche der Verbesserung oder Erneuerung von Informationssystemen scheitern an der Undurchschaubarkeit der Altsysteme. Große, monolithische Programme wechseln sich mit archaisch gewachsenen Programmkopien ab. Für die Dauer einer Neuentwicklung oder Standardsoftware-Einführung müssen die Altsysteme noch über Jahre betrieben werden. Einige stehen gar nicht zur Ablösung an. Das Modewort *Reengineering* soll nicht benutzt werden, bevor nicht folgendes geklärt ist:

- Wie kann auch in Altsystemen eine, wenn auch bescheidene, Modularität durchgesetzt werden?
- Wie läßt sich ihre Struktur transparent und damit vereinfachbar machen?

Wenn dieses in kurzer Zeit gelingt, werden *Modularität* und *Wiederverwendbarkeit* den Entwicklern gelebte Praxis und keine Schlagworte sein. Es entfällt viel unnötige Routinearbeit. Zum ersten Punkt genügte noch ein Konzept, zum zweiten bedurfte es eines Werkzeuges.

1986 war bei Vatter der Host-Rechner mit ca. 3.000 COBOL-Programmen nicht automatisch nachtlauffähig, weil ein abgebrochenes Batchprogramm die jeweilige Warteschlange blockieren konnte. Ein *condition code* wurde weder gesetzt noch ausgewertet. Das Problem wurde in drei Monaten durch *ein* Standard-Unterprogramm, die Anpassung von 700 Batch-Programmen und 300 Jobs gelöst. Die Nützlichkeit von Modulen war selbst für die Skeptiker bewiesen. Für weitere zentrale Unterprogramme, allgemeine Dateibeschreibungen in Form von Copies oder das Erkennen von Programmkopien genügte dies jedoch nicht.

Liest man von den hohen Ansprüche an ein Repository, dann sollte das folgende viel bescheidenere Werkzeug eigentlich für jeden Anwender selbstverständlich sein: Man möchte online abfragbar alle externen Referenzen zwischen Programmen, Dateien, externen Unterprogrammen, Copies, Listen und Transaktionen haben. Mit Ausnahme der Listen kann man diese Metadaten aus dem Quellprogramm fast jeder Sprache ohne Kommentar-Konventionen aus übersetzbarem Code extrahieren. In UNIX steht hierfür `awk` zur Verfügung. Abb. 3 zeigt die wichtigsten externen Referenzen eines Quellprogramms:

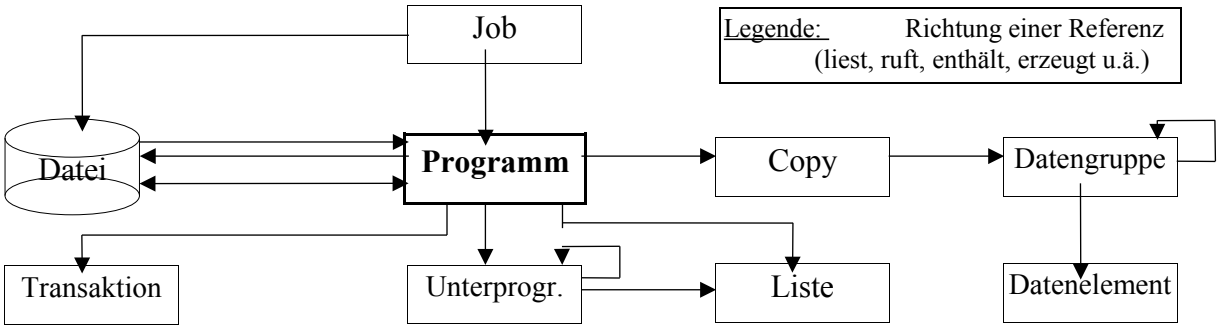

Abb. 3: Externe Referenzen von Programmen (Auswahl)

Auf einem PC wurde ein Analyseprogramm realisiert und danach auf den Produktionsrechner (IBM /VSE) portiert. Der Analysator erkennt alle erforderlichen Sprachkonstrukte von Batch- und Online-Programmen (Macro- und Command-Level für CICS). Er erzeugt je compilierbare Einheit einen Satz in einer Datenbank mit den gefundenen Referenzen. Die Analyse wird bei jeder fehlerfreien Compilation durchgeführt. Durch Einbinden in die Compile-Prozedur, die den Entwicklern nicht zugänglich ist, wird ein Umgehen der Analyse verhindert. Hierdurch kann man zu Recht von einem *aktiven* Dictionary sprechen, da die Online-Auskunft immer aktuell ist. Bedenken wegen der Laufzeit erwiesen sich selbst auf dem recht kleinen Host (6

Mips) als unbegründet. Die JCL-Analyse wurde zunächst verschoben. Ergänzt wurde später ein Abgleichprogramm der Analyse-Datenbank mit den Quell- und Objektprogramm-Bibliotheken. Der Ablauf der Compile-Prozedur zeigt die Einbindung des Analyseprogramms:

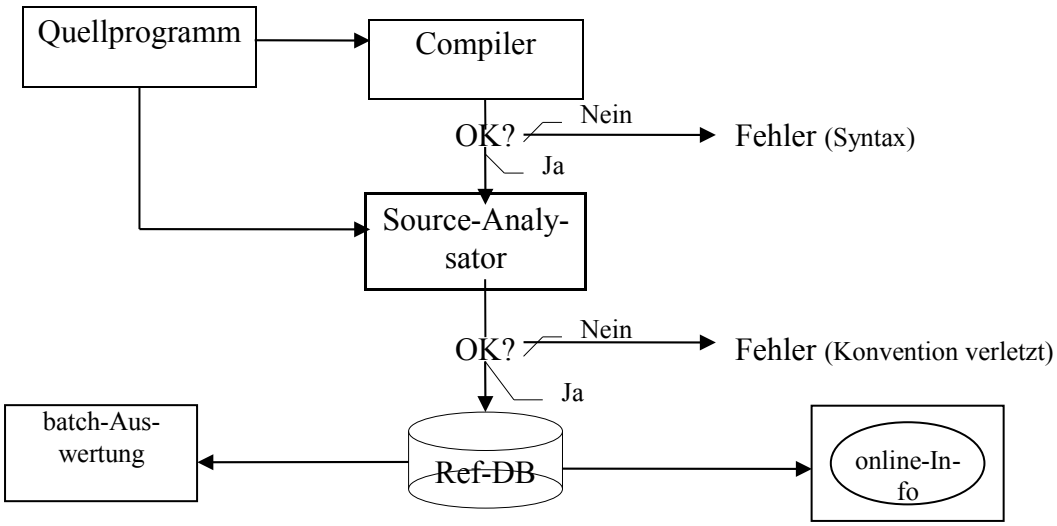

Abb. 4: Statische Source-Analyse bei jeder Compilation

Abb. 4 zeigt die Funktionalität der Source-Analyse *jedes* aktiven Dictionaries, dessen Nutzen für die Programmpflege offensichtlich ist. Man schafft durch die Transparenz Ansatzpunkte für eine Komplexitätsreduktion. So konnten durch Batch-Auswertungen viele Altprogramme kontrolliert abgelöst oder Programmkopien eliminiert werden. Daneben wurden *Daten für das Informationsmanagement* festgehalten: `Datum_letzte_Compilation` und `Anzahl_Compiles`. Hierzu ein Auszug aus den Urdaten; der Autor ist anonymisiert:

Tab. 6: Liste aus den Daten der Analyse-Datenbank, sortiert nach Anzahl Compilationen

Anz-Comp	Programm	Autor	Programm-Beschreibung	LOC
0202	Qxy195C	X	Sonderauswertungen VSKUMS	276
0188	DA0020	Z	Filtern Rechnungen DTA, SEDAS	1664
0141	DA0030	C	Erstellen SEDAS-Saetze	2029
0128	PR1014	Y	Pflege Artikelstamm alt	3042
0121	VP0287	X	Laeden nach Gebiet akt/Vorjahr	1412
:	:	:	:	:
0001	VLI667A	B	Pflege MDE-Fenster	754

Dies ermöglichte folgende Auswertung, die Basis für eine Ermittlung der Ursache häufiger Programmänderungen war. Es mußten nur wenige Programme betrachtet werden (schattiert):

AnzComp	AnzProgr
200	1
190	1
150	1
130	2
120	1
110	2
100	2
90	1
80	6
70	10
60	16
50	22
40	32
30	78
20	234
10	290
5	329
2	586
	1614

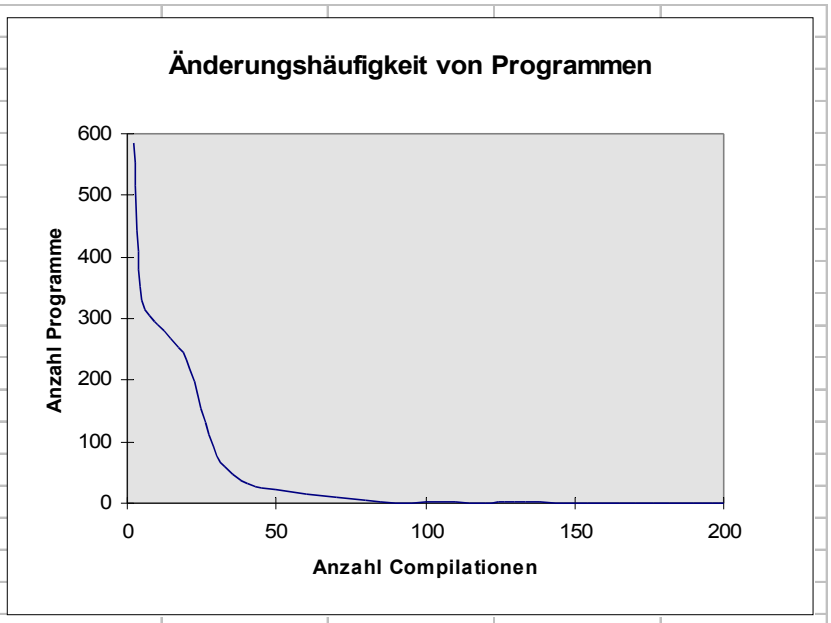

Abb. 5: Anzahl Kompilationen je Programm über ein Jahr (Werte x-Achse: Gruppen-Obergrenzen)

Es soll nicht der Eindruck erweckt werden, Anwender sollten ein solches Werkzeug selbst entwickeln. Erstrebenswerter ist sogar, es zu *kaufen*. Dem kann jedoch die mangelnde Verfügbarkeit oder Integrationsfähigkeit käuflicher Werkzeuge entgegenstehen. Z.B. wird kaum ein Anwender ein Werkzeug kaufen, für das er erst einen Compiler beschaffen muß.

Für die zweite im Konzern eingesetzte Programmiersprache (NATURAL 2) wurde das aktive Dictionary PREDICT benutzt, aus dem auch COBOL-Copies generiert wurden und *heute* auch aktive Referenzen aus COBOL-Programmen erzeugt werden können. Mengengerüste für das Informationsmanagement und brauchbare Batch-Auswertungen lieferte PREDICT allerdings 1993 standardmäßig nicht.

Die Anwendung entspricht den Thesen in folgender Weise:

Tab. 7: Umsetzung der Thesen in Beispiel 2

These Werkzeug ist ..	Umsetzung durch ..
kommunikativ	.. einheitliche Benutzung von Standardmodulen nach kontroversen Diskussionen über die „Zwangsanalyse“.
pragmatisch	.. schnellen Einsatz, zu dessen Gunsten Ergänzungen zurückgestellt worden waren (Bibliotheksabgleich, Analyse der job control).
allgemeingültig	.. Zuverlässigkeit der Referenzdaten. Sie kann ohne „Zwangsanalyse“ nicht erreicht werden.
ganzheitlich	.. Unterstützung der Wartung, u.a. bei der Ablösung komplex vernetzter Altprogramme.
akzeptierbar	.. stets korrekte online-Auskunft. Reine Batch-Auswertungen würden dagegen als Kontrollinstrument empfunden.
empirisch	.. diverse Maße für grobe Qualitätsprüfungen. Z.B. ist die Anzahl *lines of code* / *lines of comments* ein Beiprodukt der Analyse.

Der größte Nutzen des Beispiels für die Bewußtseinsbildung der Entwickler ergibt sich aus These 1: Wie immer man die methodischen Schwerpunkte des Software Engineering setzt, bleiben Quellprogramme neben den Daten die zentrale Investition eines Unternehmens im Softwarebereich. So lange es Eigenentwicklungen gibt, muß jeder Entwickler (akzeptieren)

lernen, daß er eine Unternehmensressource pflegt und keine Privatsache. Die Ressource *Programmbasis* wird durch ein aktives Dictionary unternehmensöffentlich gemacht.

6 Eine implementierte Musterarchitektur

Gute Softwarearchitekturen zu vermitteln und zu realisieren ist ein wesentliches Anliegen des Software Engineering. Es reicht jedoch nicht, sie als *Musterarchitektur* in Lehrbücher oder Richtlinien zu schreiben. Daher entstand 1984 anläßlich eines sehr großen Berichtswesen-Projektes in der Schering AG ein Softwareentwurf für Dialogsysteme, der in [Spi89, Kap. 8 u. 11] detailliert beschrieben ist. Das Gewicht der folgenden Darstellung liegt auf der Anwendung von ca. 1990 bis heute. Der Entwurf ist als Werkzeug in verschiedenen Entwicklungsumgebungen implementiert (Turbo-C, RPG III, DBASE III, MF-COBOL, NATURAL 2). Wichtiger als der Werkzeugaspekt ist jedoch die Bewußtseinsbildung bei den Entwicklern. Sie erleben vor allem in der Wartung den Nutzen und die Releasefähigkeit einer Standardarchitektur. Sie entlastet von unnötiger Routinearbeit.

Die Grundidee einer wiederverwendbaren Musterarchitektur ist einfach und auch Basis objektorientierter Systeme: Man isoliere anwendungsunabhängige von anwendungsspezifischen Funktionen und Datentypen und notiere den Entwurf in einer implementierungsneutralen Form. Die Implementierung kann je nach Funktionalität des Basissystems partiell aber nicht grundsätzlich vom Entwurf abweichen, indem man Operationen des Basissystems übernimmt („erbt“) oder selbst schaffen muß.

Die Lösung ist eine Mischung aus generierbaren Programmen, Programmskeletten und universellen Standardmodulen. Die Programme kommunizieren über eine standardisierte Schnittstelle. Mit dem Werkzeug kann man nach dem Entwurf anwendungsspezifischer Masken oder Listen in sehr kurzer Zeit einen lauffähigen Prototyp der Benutzeroberfläche erzeugen, der stufenweise zum Produktivsystem ausbaubar ist. Die Musterarchitektur umfaßt ein Schichtenmodell, das sich auch ohne Client-Server-Hardware realisieren läßt, indem bei der Modularisierung zwischen *Präsentation*, *Anwendungskern* und *Datenbasis-Zugriff* unterschieden wird. Abb. 6 zeigt diesen Zusammenhang; die Schichten sind gestrichelt angedeutet.

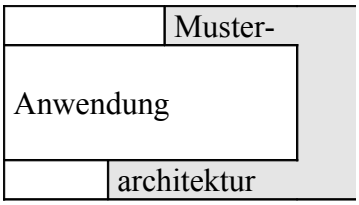

Abb. 6: Konzept einer Musterarchitektur

Der schattierte Teil des Softwaresystems ist wiederverwendbar. Die häufigste Anwendung einer Musterarchitektur ist ein Dialogsystem, bei dem man Benutzeroberfläche und Dialogsteuerung standardisiert. Das Werkzeug muß mehr können als eine grafische Oberfläche, vor allem bezüglich der dynamischen Steuerung und der Behandlung von Systemfehlern [vgl. hierzu auch Den91, Kap.10]. Die Oberfläche impliziert vor allem eine wichtige Entwurfsentscheidung zur Modularisierung: Da die Steuerung aus den anwendungsspezifischen Modulen ausgelagert wird, steht hinter jeder Maske und jedem Fenster immer nur genau ein Modul. Die Standardisierung ergibt bestimmte Typen anwendungsspezifischer Module, z.B. *Eingabedaten-Prüfung* oder *Bildschirmanzeige*. Die Gefahr gravierender Entwurfsfehler wird mit einem solchen Werkzeug auf den Datenbankentwurf begrenzt.

Die Leistungen des Systems lassen sich schlagwortartig anhand der wichtigsten Standard-Module aufzählen:

- Dialogstack
- TAC-Handler (Transaktionscode)
- Error-Handler/Dialog
- Init-Programm
- Hilfe-Handler
- PF-Tasten-Handler
- Message-Handler
- Error-Handler/Batch
- Job-Control-Generator
- Kompaß (wo bin ich im Dialogbaum?)

Dazu existieren ein Batch- mehrere Dialogskelette. Batchprogramme können unter bestimmten Bedingungen aus dem Dialog uhrzeitgesteuert gestartet werden, indem ein Job generiert wird.

Auf Skelettprogrammen beruhende Generatoren gibt es bei vielen Anwendern und Softwarehäusern. Alle dem Autor bekannten Lösungen basieren aber nicht auf einer dokumentierten Architektur, sondern auf einer konkreten Implementierung. Dies verhindert eine Wiederverwendung bei Plattformwechseln oder grundlegenden Neuentwicklungen. Die hier beschriebenene Lösung konnte über mehrere Generationen von Plattformen weiterentwickelt werden. Den Entwicklern ist sie als *Konzept* bekannt und nicht nur als isolierte Implementierung. Sie haben über mehrere Releases erfahren, welchen praktischen Wert eine durchdachte Software-Architektur seit nunmehr 10 Jahren haben kann, die im ursprünglichen Entwurf erst später verfügbare Leistungen der Basissysteme vorwegnahm. Die bisherigen Releases für die am stärksten benutzte Implementierung in NATURAL haben z.B. folgende Stufen durchlaufen:

Tab. 8: Evolution der NATURAL-Implementierung

Jahr	Implementierung	Plattform
1985	NATURAL 1.2 mit COBOL-Unterprogrammen	IBM-Host
1990	NATURAL 2.1 in homogener, modularer Struktur	IBM- und SNI-Host
1991	Einbindung Dictionary PREDICT für Hilfetexte	siehe oben
1995	NATURAL 2.2 als Client-Server-Anwendung	Host (auch UNIX) mit Windows-Client

Man beachte, daß NATURAL 1 und NATURAL 2 nur dasselbe Marketing-Etikett tragen. Es handelt sich um zwei weitgehend verschiedene Programmiersprachen. Durch die Architektur und Einheitlichkeit der Programmskelette konnte die Release-Umstellung programmgestützt geschehen: Modifikation der NATURAL-1-Programme und Neucompilation. Daß man Programme automatisch verändern kann, war für die Entwickler eine neue Erfahrung.

Ein solches System hat erhebliche Wirkungen auf Mitarbeiter-Qualifikation und Produktivität. Bei Vatter wurden mangels verfügbarer Standardsoftware (variantenreiche und modische Massenfertigung) die gesamten logistischen Anwendungen nach diesem Konzept in relativ kurzer Zeit neu erstellt und erheblich erweitert [Spi93]. Dabei wurde auch ein verteiltes PPS-System entwickelt, bei dem die Werkstatt-Komponente auf einem PC-Netz implementiert war. Dieses Teilsystem war in der DBASE-Version der Musterarchitektur realisiert, da ADABAS und NATURAL 1992 noch nicht auf kleineren Systemen zur Verfügung standen.

Ohne die hohe Produktivität der Mitarbeiter und Werkzeuge wäre eine so umfangreiche und technisch riskante Entwicklung nicht möglich gewesen. Die Entwickler hatten den „Kopf frei“ für die Anwendungen und brauchten sich nicht um ständig wiederkehrende Probleme wie Dialogaufbau und -steuerung zu kümmern. Der Entwicklungsaufwand betrug für einen Teil der Neuentwicklungen bis Mitte 1992 44,3 MJ für 1021 Module, das sind ca. 70 Stunden/Modul über alle Entwicklungsphasen. Daß die Produktivität keine scheinbare mit schneller Entwicklung und dann aufwendiger Wartung war, läßt sich anhand der äußerst geringen Wartungsaufwände aus der Entwicklungs-Datenbank nachvollziehen. Das erste größere System (1988, Tourenplanung Außendienst) benötigte in den ersten 3½ Jahren seines Einsatzes nur 706 Stunden für Wartung, fast durchweg für funktionale Erweiterungen, das sind nur ca. 4,5 MM.

Was sind nun die Entsprechungen der Architektur mit unseren Thesen?

Tab. 9: Umsetzung der Thesen in Beispiel 3

These Werkzeug ist ..	Umsetzung durch ..
kommunikativ	.. einheitliche und für alle durchschaubare Software. Selbst Teams, die in verschiedenen Sprachen implementieren (müssen!), verstehen einander.
pragmatisch	.. eine erste Implementierung in NATURAL 1 trotz erheblicher Mängel dieser 4GL, da der Nutzen überwog.
allgemeingültig	.. die Verbindlichkeit der Architektur für alle Dialog-Neuentwicklungen unabhängig von der Plattform (Host oder dezentrales System).
ganzheitlich	.. Einheitlichkeit und Modularität. Dies bringt vor allem in der Wartung Nutzen, etwa bei der Erweiterung der Hilfe-Funktion durch nachträgliches Einbinden des Dictionary.
akzeptierbar	.. allgemeine Anerkennung des Systems.
empirisch	.. die Möglichkeit, im Init-Programm je Applikation *Nutzungsdaten* festzuhalten (User-neutral!). Daneben kann man automatisch Übersichten wie etwa Modul-Bäume oder Benutzungshäufigkeiten von Modulen gewinnen.

Das Beispiel zeigt, wie man auch komplexe Konzepte des Software-Engineering werkzeuggestützt vermitteln kann und dies mit den heute besonders dringenden Produktivitätssprüngen verbindet. Die Entwickler erfahren vor allem die Investitionssicherheit guter Konzepte. Dies ist in der gegenwärtig äußerst schnellebigen Technologielandschaft nicht selbstverständlich.

Wenn Anwender heute noch Eigenentwicklungen betreiben, *müssen* sie solche Werkzeuge haben, um dem Legitimationsdruck gegenüber wertschöpfenden Abteilungen standzuhalten. Werkzeuge dieser Art sind für erfolgreiche Standardsoftwarehersteller längst selbstverständlich, natürlich in sehr viel differenzierterer Form als hier dargestellt.

7 Schluß

Es ist nicht besonders schwierig, wiederverwendbare Module oder Klassen zu *schreiben*. Schwieriger - und nur dies fördert Wirtschaftlichkeit - ist es, diese Module und Klassen zu *finden*, ihre Wirkungen zu verstehen und dann tatsächlich *wiederzuverwenden*. Alle drei Beispiele handeln von der Wiederverwendung von Komponenten und Konzepten als einem besonders zentralen Anliegen des Software Engineering:

- Beispiel 1 von den kleinsten Elementen einer Datenbasis und deren disziplinierter Konstruktion und Benutzung;
- Beispiel 2 von Modulen als größeren Einheiten, die selbst im Nachhinein eine Programmbasis ein wenig verbessern, vor allem aber Entwicklermentalitäten verändern können;
- Beispiel 3 von der Wiederverwendung einer Architektur, die aus vielfach benutzten Modulen, Datenstrukturen und Abläufen besteht.

Alle drei Beispiele zielen auf eine werkzeuggestützte Kommunikation über Ideen und Konzepte und damit die Überwindung individualistischen Denkens. Rein workstation-orientierte Werkzeuge ohne Entwicklungs-Datenbank (Einzelplatz-Systeme) fördern genau dieses Anliegen *nicht*.

Wiederverwendung als konzeptionellen roten Faden halte ich für den wichtigsten Gedanken, der das Handeln von Entwicklern in der kommenden Welt verteilter Hardware- und Softwaresysteme bestimmen sollte. In dieser Welt *gewinnt* man ohne Zweifel Flexibilität. Unvorbereitete Entwickler könnten aber auch leicht den Überblick *verlieren*.

Danksagung. Ich danke den Gutachtern und Herrn Manfred Mönckemeyer, Schering AG Berlin, für wertvolle Hinweise.

Literatur

[**Boe87**] *Boehm, B.W.:* Improving Software Productivity. In: IEEE Computer 20(1987) 9, 43-57.

[**Den91**] *Denert, E.*: Software-Engineering. Springer, Berlin et al. 1991.

[**Gra93**] *Grams, T.*: Täuschwörter im Software Engineering. In: Informatik-Spektrum 16(1993) 3, 165-166.

[**Hanf91**] *Hanf, V.*: Integrierte Datenmodellierung bei BMW - ein Erfahrungsbericht. In: Wirtschaftsinformatik 33(1991) 4, 300-307.

[**HeFr94**] *Hesse, W.; Frese, M.*: Zur Arbeitssituation in der Software-Entwicklung. Resümee einer empirischen Untersuchung. In: Informatik Forschung und Entwicklung 9(1994) 4, 179-191.

[**Ort94**] *Ortner, E.*: KASPER - Ein Projekt zur natürlichsprachlichen Entwicklung von Informationssystemen. In: Wirtschaftsinformatik 36(1994) 6, 570-579.

[**Spi83**] *Spitta, Th.*: Eine verteilte Projektbibliothek in Verbindung mit einem Data Dictionary. In: Softwaretechnik-Trends 3(1983) 2, 83-103.

[**Spi89**] *Spitta, Th.*: Software Engineering und Prototyping. Springer, Berlin et al. 1989.

[**Spi93**] *Spitta, Th.*: Sechs Jahre Anwendungsentwicklung mit Prototyping - Revision von Begriffen und Konzepten -. In: Züllighofen, H. et al. (Hrsg.): Requirements Engineering '93, Teubner, Stuttgart 1993, 49-66.

[**Spi96**] *Spitta, Th.*: Wiederverwendbare Attribute als Ordnungsfaktor der Unternehmensdaten - Konzept und empirische Analyse -. Erscheint in: Ortner, E.; Schiennmann, B.; Thoma, H. (Hrsg.): Natürlichsprachlicher Entwurf von Informationssystemen, Workshop Mai ‘96 Tutzing, Universitätsverlag, Konstanz 1996, 14 S.

[**Vett82**] *Vetter, M.*: Aufbau betrieblicher Informationssysteme mittels konzeptioneller Datenmodellierung. Teubner, Stuttgart 1982 (7.Aufl. 1991).

[**Wall90**] *Wallmüller, E.*: Software-Qualitätssicherung in der Praxis. Hanser, München - Wien 1990.

[**WeOr92**] *Welz, F; Ortmann, R.*: Das Softwareprojekt - Projektmanagement in der Praxis. Campus, Frankfurt - New York 1992.