Software Engineering und CASE – Begriffserklärung und Standortbestimmung

Software Engineering and CASE – what it means and where we are

Jochen Ludewig, Institut für Informatik der Universität Stuttgart

Jochen Ludewig, Studium der Elektrotechnik an der TU Hannover, Aufbaustudium Informatik an der TU München. Mitarbeiter am Institut für Datenverarbeitung des Kernforschungszentrums Karlsruhe, 1981 Promotion durch die TU München. Leiter des Projekts Software Engineering im Brown Boveri Forschungszentrum in Baden/Schweiz. Seit 1988 Prof. für Software Engineering an der Universität Stuttgart.

CASE-Tools werden heute als wichtige Mittel der Leistungs- und Qualitätssteigerung im Software Engineering betrachtet. Diese Einschätzung ist richtig, wenn sie mittel- und langfristig verstanden wird; sie ist falsch, wenn man erwartet, rasche Hilfe zu bekommen, die Versäumnisse in der Methodik und Schulung ausgleicht.

Die heute angebotenen Werkzeuge weisen charakteristische Mängel auf, die – entgegen den Ankündigungen – ihren durchgehenden Einsatz sehr schwer machen. Trotzdem kann unter bestimmten Voraussetzungen, auch organisatorischen, die Qualität des Entwicklungsprozesses tatsächlich erhöht werden. Diese Verbesserung wirkt sich auch auf die Produktivität aus.

CASE tools have given rise to expectations of improved productivity and quality in software engineering. In fact, their medium- or long-term effect will be positive, but CASE tools cannot serve as a handy compensation for a lack of methods and training.

Tools which are now available suffer from typical deficiencies which – contrary to the announcements – practically prevent their continious application in the software life cycle. Nevertheless, under certain technical and organisational conditions, a better quality of the process of software development can be achieved, which in turn will influence the productivity.

Einleitung

In der Hardware gab es (und gibt es voraussichtlich weiterhin) über viele Jahre hinweg *exponentielle Verminderung* von
- Größe (Chipfläche pro Bauelement)
- Zyklus- und Operationszeiten
- Preis für einen logischen Baustein, z.B. ein Speicherbit.

Dagegen haben wir bei der Software seit langem im wesentlichen *konstante Kosten pro Zeile Programmcode*. Der Grund ist sehr einfach: Die menschliche Intelligenz (d.h. die geistige Leistung) läßt sich kaum verändern. Darum ist eine Verbesserung (fast) nur durch *effizienteren* Einsatz der Denkleistung möglich. Was wir nicht vermehren können, müssen wir so sorgfältig wie möglich einsetzen.

An dieser Stelle betritt der junge Held die Bühne: CASE.
Seine Verheißung ist die **Steigerung**
- **der Produktivität**
- **der Produkt-Qualität** (z.B. Senkung der Fehlerhäufigkeit)
- **der Projekt-Qualität** (z.B. Einhaltung des Projektplans).

In diesem Artikel soll versucht werden, den Begriff CASE vor dem Hintergrund des Software Engineerings genauer zu fassen und festzustellen, welchen Stand das „Computer Aided Software Engineering" heute hat.

Definitionen

„Software Engineering" und einige verwandte Begriffe sind in IEEE-Std 729-1983 definiert:

software engineering. The systematic approach to the development, operation, *maintenance,* and retirement of *software.*

software development process. The process by which user needs are translated into *software requirements,* software requirements are transformed into *design,* the design is implemented in code, and the code is tested, documented, and certified for *operational* use.

Im Zusammenhang dieses Beitrags spielen die Begriffe *Methode, Sprache* (oder *Notation*) und *Werkzeug* wichtige Rollen:
- Eine **Methode** hat den Charakter einer Handlungsanweisung, also eines Regelwerks oder Rezepts
- eine **Sprache** oder **Notation** ist eine Konvention für die Kommunikation; Beispiele sind Deutsch, Esperanto, Ada, Datenflußdiagramme
- ein **Werkzeug** speichert Information oder formt sie um (insbesondere auch in eine andere Sprache). Beispiele: Tonband, Brille, File-System, Compiler.

Methode(n), Sprache(n) und Werkzeug(e) bilden ein **System**, wenn sie durch das Substrat gemeinsamer **Konzepte** verbunden sind. Entsprechend lassen sich die Begriffe in Dreiecksform anschaulich darstellen (Bild 1).

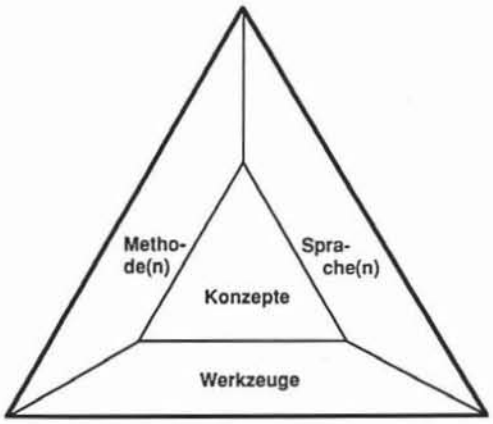

Bild 1: Das System-Dreieck

„CASE" kommt in der IEEE-Begriffsnorm noch nicht vor, der Ausdruck ist jünger und durch den freizügigen Gebrauch in der Werbung eher verschwommen. Wir finden aber als verwandte Begriffe:
- *program synthesis.* The use of *software tools* to aid in the transformation of a *program specification* into a *program* that realizes that *specification*
- *programming support environment.* An integrated collection of *tools* accessed via a single *command language* to provide programming support capabilities throughout the *software life cycle.* The environment typically includes tools for designing, editing, compiling, loading, *testing, configuration management,* and project management
- *software tool.* A *computer program* used to help develop, test, analyze, or maintain another computer program or its *documentation;* for example, *automated design tool, compiler,* test tool, *maintenance tool.*

Die Definitionen zahlreicher Begriffe beginnen mit der Charakterisierung „A software tool that ..." oder „A class of software tools ...":

automated design tool, automated test generator, automated verification system, automated verification tools, comparator, design analyzer, dynamic analyzer, instrumentation tool, software monitor, standards enforcer, static analyzer, timing analyzer.

CASE ist eine seit etwa 1985 gebräuchliche Abkürzung für „Computer Aided Software Engineering". Ich schlage als pragmatische Definition vor:

CASE. die Bearbeitung von Software mittels Werkzeugen (den CASE-Tools), die auf Rechnern realisiert sind und vom Menschen direkt gesteuert werden.

CASE-Tools sind also Programme, die der Bearbeitung von Software-Komponenten, also auch anderen Programmen, dienen. Beispiele sind Editoren, Datei-Systeme, Testmonitore, Spezifikationssysteme und Dokumenten-Generatoren. Ausgeschlossen sind durch die Definition dagegen beispielsweise
- NC-Programme, weil sie nicht der Bearbeitung von Software dienen
- Programmiersprachen, weil diese nicht zu den Werkzeugen zählen
- eine Kartei mit Angaben über wiederverwendbare Module, weil sie nicht auf dem Rechner realisiert ist
- der Scheduling-Mechanismus eines Time-Sharing-Systems, weil er nicht direkt vom Benutzer gesteuert wird.

Eine weitere Klassifizierung von CASE-Tools ist problematisch. Die Unterscheidung von „Upper CASE" und „Lower CASE" klingt auf den ersten Blick plausibel, wenn man damit die Werkzeuge für die frühen Phasen (Analyse, Spezifikation und Entwurf) von denen für die Bearbeitung von Code (Compiler, Linker, Loader, Debugger) unterscheidet. Unklar ist dann aber die Einordnung der Werkzeuge für die späteren Phasen, z.B. für die Integration. Im Sinne des von oben nach unten dargestell-

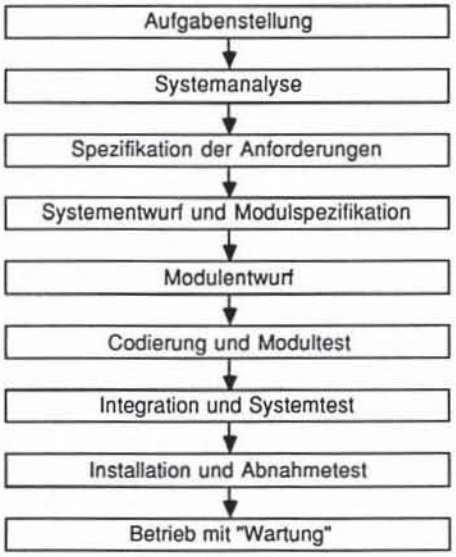

Bild 2: Ein Phasenplan

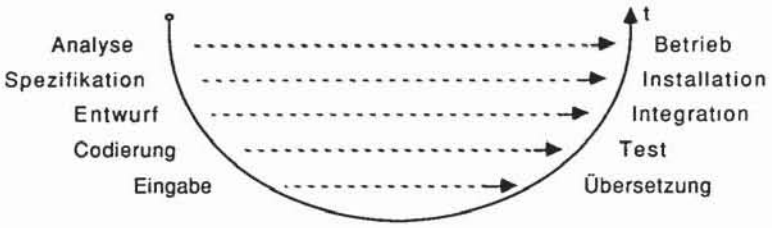

Bild 3: Die Badewannen-Kurve

ten Phasenplans (Bild 2) ist dies offenbar „Lower CASE".

Auf die Abstraktionsebene bezogen (Bild 3 aus Frühauf, Ludewig, Sandmayr, 1991) gehört die Integration aber zum „Upper CASE".

Im gängigen Sprachgebrauch bezeichnet CASE nur diejenigen Werkzeuge, die erst im Verlauf der letzten zehn Jahre entstanden sind. Werkzeuge für die Bearbeitung von Programmcode (wie Compiler, Debugger, auch Dateisysteme) sind damit implizit ausgeschlossen, sie haben die neue Fahne auch nicht nötig. Man kann CASE also auch so definieren:

CASE. die Verwendung von Software-Werkzeugen im Rahmen der Software-Bearbeitung für Tätigkeiten, die man bis 1970 sicher von Hand ausführen mußte.

Der für CASE wesentliche Aspekt der Integration wird unten diskutiert.

Im heutigen Sprachgebrauch wird „CASE" oft weiter eingeschränkt, nämlich auf Werkzeuge für die frühen Phasen.

Methoden und Werkzeuge

Die Bewertung von Methoden und Werkzeugen ist unterschiedlich je nach Perspektive:

Dem Unternehmen liegt vor allem an den Methoden. Diese steigern die Qualität und verbessern die Zusammenarbeit, führen also zu niedrigeren Kosten. Der Schulungs- und Einführungsaufwand und das Risiko des Scheiterns sind dadurch in der Regel zu rechtfertigen. Für den Mitarbeiter ist die Methode dagegen vor allem eine Bedrohung, denn sie bringt neue Anforderungen und die Gefahr des Versagens mit sich. Daher stoßen neue Methoden in aller Regel auf großen Widerstand.

aus Sicht	bewirken moderne Methoden
des Unternehmens	Schulungsaufwand, dann Steigerung verschiedener Qualitäten, dadurch Senkung des Aufwands, d.h. der Kosten
des Entwicklers	Verbesserung der Qualifikation, aber auch Zwang zur Umstellung, Risiko des Versagens

Bei den Werkzeugen verschiebt sich die Wertung: Hier hat nicht nur das Unternehmen Vorteile (Produktivität), sondern – nach Überwindung der unvermeidlichen Schwellenangst – mindestens ebensosehr der Mitarbeiter, der von banalen Aufgaben der Informationsverwaltung und -prüfung entlastet wird und ein interessantes Spielzeug bekommt.

aus Sicht	bewirken Werkzeuge (CASE-Tools)
des Unternehmens	Evaluations- und Beschaffungskosten, dann aber Verbesserungen der Dokumentation, Kontrolle, Produktivität
des Entwicklers	Einarbeitung, dann Entlastung von trivialen Arbeiten, Erfolgserlebnisse, bessere Sichtbarkeit der Leistungen

Die unterschiedliche Bewertung, die in den beiden Gegenüberstellungen zum Ausdruck kommt, hat eine interessante Konsequenz, die sich in vielen Erfahrungsberichten etwa wie folgt niederschlägt:

„Das Wichtigste war die *Einführung* des Werkzeugs; sie war mühsam. Sein Gebrauch hat *dann* aber zu Verhaltensänderungen geführt, die sich sehr vorteilhaft auswirken. *Jetzt* könnten wir eigentlich auf das Werkzeug verzichten."

Daher sollten Werkzeuge und Methoden nicht nur gut aufeinander abgestimmt sein, sondern stets als Paketlösung ausgewählt und eingeführt werden.

Die Entwicklung von CASE

Software-Engineering-Werkzeuge hat es natürlich schon lange vor dem Wort CASE gegeben, nicht nur für die Code-Bearbeitung, sondern auch für die frühen Phasen. Ein Beispiel ist das System PSL/PSA (Teichroew, Hershey, 1977), das bereits 1969 (!) konzipiert worden war. Anfangs der 80'er Jahre war „Software Engineering Environment" das entsprechende Schlagwort (Hünke, 1981).

In den letzten zehn Jahren hat sich aber einiges getan:

■ Die Benutzerschnittstelle ist heute ganz dramatisch verbessert. Die Teletype-Schnittstelle (zeilenorientierte Ein- und Ausgabe, im übrigen Kommunikation über Dateien) wurde ersetzt durch eine graphische Schnittstelle mit Multi-Windowing. Dies war möglich durch die Verfügbarkeit schneller Workstations mit hochauflösendem Bildschirm. Damit kann man heute Informationen nach freier Wahl in linearer Form (d.h. durch Text) oder direkt in graphischer Form ein- und ausgeben, obwohl sie intern als logisches Netzwerk gespeichert wird, so

daß verschiedene Prüfungen und Umformungen vorgenommen werden können
- die Notwendigkeit einer Integration der Werkzeuge für verschiedene Phasen ist allgemein anerkannt und teilweise berücksichtigt. Der Werbung zufolge ist sie mit vielen Werkzeug-Systemen auch bereits erreicht, doch leider bleibt die Realität weit dahinter zurück. Die Schwierigkeit liegt nicht zuletzt in der Datenschnittstelle (siehe unten)
- durch die Standardisierung bei der (virtuellen) Basismaschine wurde der Markt wesentlich größer. Fand man vor zehn Jahren mit viel Glück gerade *ein* Werkzeug, das auf einer bestimmten Maschine und unter einem bestimmten Betriebssystem lief, so gibt es heute viele, die auf UNIX, DOS oder OS/2 abgestimmt sind und damit auf sehr unterschiedlicher Hardware eingesetzt werden können
- die daten*fluß*orientierten Entwicklungstechniken (SADT, Ross, 1985, vor allem aber Structured Analysis, deMarco, 1978; McMenamin, Palmer, 1988) haben sich neben den daten*struktur*orientierten (Jackson, 1975; Cameron, 1983) weit verbreitet und werden durch entsprechend viele Werkzeuge unterstützt.

Klassifikationen und Entwürfe für eine Taxonomie

Überblick setzt Struktur voraus; eine Klassifikation der Werkzeuge ist aber wegen ihrer vielen Dimensionen schwierig, wie der mißlungene Versuch von Dart et al. (1987) zeigt.

Life-Cycle-bezogene Klassifikation

Die Werkzeuge (oder bei integrierten Werkzeugen ihre Komponenten) lassen sich einteilen in solche, die bestimmten Phasen (oder genauer: Tätigkeiten) zugeordnet werden können, und in andere, die auf mehreren oder allen Ebenen zum Einsatz kommen.

Universelle Werkzeuge
Editoren
Datei-Systeme
Datei-Vergleicher
Datenbanken und Data Dictionary-Systeme
Werkzeuge für das Configuration Management
Report-Generatoren

Werkzeuge für spezifische Phasen
Spezifikationssysteme
Entwurfswerkzeuge
Code-Generatoren
Testtreiber, Testinstrumentierer
Werkzeuge zur Verwaltung von Änderungen

Klassifikation nach Struktur und Mächtigkeit

Diese Klassifizierung stammt aus einem Tutorium von Riddle (1985).

Basic Environment
Editor, File System, ...
Tool Boxes (kombinierbare Werkzeuge)
UNIX
Information Repositories
Data Dictionary
Support Systems
PSL/PSA und andere
Method oriented Environments
mit gewissen Einschränkungen Systeme wie Promod, TeamWork, StP, IEW und andere

Klassifikation von Werkzeug-Ausstattungen

Die folgende Liste entstand auf einer ACM-Tagung (Howden et al., 1982).

Feigenblatt
- einfache, manuelle Spezifikationsmethode
- Datenbanksystem für den Entwurf
- Versionen-Verwaltung für den Quellcode
- Datei-Vergleicher für die Validierung

Leopardenfell (auf Deutsch besser: **„Lumpenmantel"**)
wie oben, aber zusätzlich
- einfaches Projektdatenbanksystem
- einfaches Werkzeug zur Analyse der Zusammenhänge in der Datenbank
- halbformales Spezifikationssystem wie PSL/PSA
- Methodenorientiertes Entwurfswerkzeug
- automatische Konfiguration, automatischer Test
- System zur Projektüberwachung

Overall
wie oben, aber zusätzlich
- Integration der Werkzeuge
- zusätzliche Werkzeuge, vor allem für Prüfungen und Debugging

Raumanzug
wie oben, aber vollständig auf einem Datenbanksystem integriert

Tatsächlich verfügbar sind diejenigen Werkzeuge, die zum Feigenblatt oder zum Leopardenfell gehören. Der Overall ist „leading edge" und auf dem Markt nicht verfügbar, der Raumanzug bleibt vorläufig Utopie. Die Industrie lebt nach meiner Kenntnis auch heute noch überwiegend im paradisischen Zustand, denn sie verzichtet selbst auf das Feigenblatt.

Architektur offener Werkzeugsysteme

An ein CASE-Tool werden Anforderungen gestellt, die seine Architektur weitgehend festlegen:
- Verschiedene Funktionen sollen zur Verfügung stehen, damit während der ganzen Entwicklung (oder doch wenigstens in mehreren Phasen) Unter-

stützung geboten werden kann. Die Menge dieser Funktionen muß offen bleiben, weil wir heute nicht genug wissen, um einen abgeschlossenen Werkzeugkasten zu definieren (**„offene Funktionsvielfalt"**)
- der Benutzer möchte nicht die Vielfalt dieser Teile sehen, sondern eine einheitliche Bedienschnittstelle (**„uniforme Oberfläche"**)
- die Werkzeuge müssen ohne weiteres Zutun der Benutzer direkt miteinander kommunizieren (**„Integration"**).

Damit entsteht das „Software-Rack" (Bild 4): Zwischen uniformer Frontplatte (Bedienschnittstelle) und Backplane Bus (Software-Engineering-Datenbank) liegen die verschiedenen Werkzeuge.

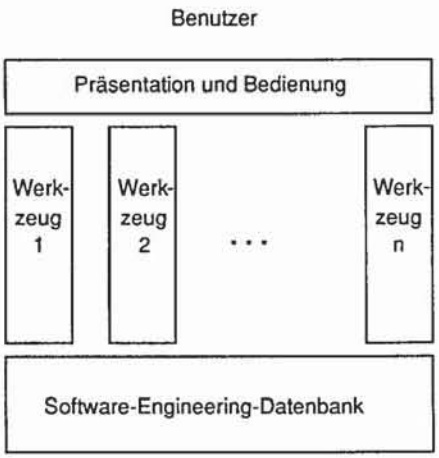

Bild 4: Architektur eines integrierten Software-Werkzeugs

Syntaktische und semantische Werkzeugintegration

Während an der Oberfläche heute jeder beliebige Komfort verlangt und geboten wird, ist die Entwicklung auf der „Rückseite" unserer Systeme viel weniger weit. Betrachten wir dazu, wie die Probleme beim Übergang von einfachen, „freistehenden" Werkzeugen zu integrierten Entwicklungsumgebungen wachsen:

Simple Werkzeuge, die als Transformatoren arbeiten, machen keine Schwierigkeiten. Beispiel: ein Pretty Printer liest Quellcode aus einer Datei und erzeugt formatierten Quellcode in einer zweiten.

Werkzeuge, die verkettet sind, erfordern dagegen bereits präzise Konventionen über die Form der zu übergebenden Information. Solche Absprachen sind aber noch immer relativ leicht zu treffen, weil sie nur bilateral binden. Beispiel: Ein Compiler erzeugt Objectcode, der vom Linker verarbeitet wird.

Werkzeuge mit Interaktion haben neben der Datenschnittstelle auch die Schnittstelle zum Benutzer. Beispiel: ein Editor bearbeitet eine Datei, geführt durch einen Benutzer.

Das integrierte Werkzeugsystem, wie es in Bild 4 gezeigt wurde, stellt bezüglich Schnittstellen die höchsten Ansprüche, und dies ist wohl der Grund, warum auf diesem Gebiet die Entwicklung wesentlich langsamer verlaufen ist, als vor gut zehn Jahren erwartet wurde.

Für die technische Realisierung der Software-Engineering-Datenbank gibt es verschiedene Lösungen, die nachfolgend verglichen werden sollen.

Informationen, die zu der zu entwickelnden Software gehören, können gespeichert werden:
a) informal (d.h. als natürlichsprachlicher Text) in Textdateien
b) (halb-)formal (d.h. in einer Programmiersprachen-ähnlichen Notation) in Textdateien
c) (halb-)formal in Textdateien, die durch formale Schnittstelleninformationen ergänzt sind
d) in einer Datenbank (die die Verwaltung umfangreicher Informationen, z.B. Modul-Beschreibungen in natürlicher Sprache, an ein Dateisystem delegiert).

Diese Lösungen stellen unterschiedliche Anforderungen an die Werkzeugausstattung:
a) erfordert nur Dateisystem und Editor
b) zusätzlich ein Werkzeug, das die Querverweise analysiert (Cross Reference Tool)
c) zusätzlich eine Schnittstellenverwaltung und Änderungskontrolle
d) ein Datenbanksystem mit Abfragesprache.

Diese Lösungen geben dem Anwender unterschiedlich gute Unterstützung bei der Verwaltung seiner Informationen. Als Beispiel sei angenommen, daß die Beschreibung eines großen Software-Systems die folgenden Informationen verstreut und im Fall γ nur implizit enthält:
α) Modul A verwendet F1 aus B
β) Modul B stellt F2 und F3 zur Verfügung
γ) F3 wird nirgends verwendet.

Es besteht also ein Widerspruch zwischen den beiden Aussagen α und β, während γ ein Hinweis darauf ist, daß das System überflüssige Teile enthält oder sonst ein Mangel vorliegt. In der Terminologie der Compiler erwarten wir im ersten Fall eine Fehlermeldung, im zweiten eine Warnung.

Lassen sich diese Inkonsistenzen auch mit den Werkzeugen a bis d entdecken?
a) gibt uns keine Chance. Die Prüfung von Texten in natürlicher Sprache ist heute (und vermutlich auch in absehbarer Zukunft) nicht möglich
b) erfordert die Prüfung *aller* Dateien, was sehr viel Zeit erfordert und daher praktisch nur in der Mittagspause oder über Nacht möglich ist, also offline, nicht interaktiv
c) erfordert nur die Analyse geänderter Dateien und den Abgleich der Schnittstelleninformationen, was auch interaktiv geht, wenn man eine kurze Wartezeit akzeptiert

d) erlaubt die Prüfung anläßlich jeder atomaren Operation, also nicht erst, wenn eine geänderte Datei abgespeichert wird.

Aus diesen Gründen ist die Lösung d die komfortabelste. Leider stellt sie auch die höchsten Anforderungen an die Abstimmung der verwendeten Werkzeuge, so daß sie in der Praxis noch kaum anzutreffen ist. Einer der Gründe ist die Ausrichtung heute üblicher Datenbanksysteme auf große Mengen homogener, relativ kleiner Informationsbausteine, die in schnellen Transaktionen bearbeitet werden; für unsere Zwecke benötigen wir ganz andere Konzepte (sog. *Non-Standard Database Systems*). An diesem Problem wird derzeit intensiv gearbeitet (vgl. Abramowicz et al., 1988; Matheis, 1990).

Schwerer wiegt aber das – durch Datenbanksysteme nicht behebbare – Fehlen eines Datenmodells, das sich für alle Werkzeuge eignet. Objektorientierte Datenbanken könnten hier eine Etikettierung der abgespeicherten Informationen erleichtern, aber in jedem Fall ist ein plausibles Modell nötig, das heute niemand kennt.

Durch das Geschick der Werkzeuganbieter, für die skizzierten Speicherungsprobleme Lösungen zu finden, dürfte in Zukunft die Grenzlinie zwischen brauchbaren und anderen Werkzeugen definiert sein.

Das Problem der vertikalen Integration

Die oben beschriebene Integration auf einer gemeinsamen Datenbank kann man als **horizontale Integration** charakterisieren: Die Werkzeuge stehen nebeneinander, nur ihr Zweck impliziert eine bestimmte Reihenfolge der Anwendung; beispielsweise wird ein Entwurf erst in Text-Form eingegeben, dann geprüft und schließlich graphisch ausgegeben.

Die **vertikale Integration** bezeichnet hier die Herstellung eines logischen, damit auch von den Werkzeugen erkannten Zusammenhangs zwischen den verschiedenen Dokumenten, die im Laufe der Entwicklung entstehen.

Dieses Problem hat sich bisher als die härteste Nuß erwiesen, und die Lösungsansätze sind noch sehr mager: Schon die Generierung eines Dokuments aus einem logisch vorangehenden (also z.B. die Codeerzeugung aus dem Entwurf) ist kaum möglich. Wo sie angeboten wird, handelt es sich entweder um eine **Skelett-Generierung**, bei der ein vom Programmierer auszufüllender Programm-Rahmen entsteht, oder um die **Baustein-Konfiguration**, bei der der Programmierer die Inhalte schon auf Entwurfsebene in Code-Form vorgibt; dieser Code kann dann durch die Generierung konfiguriert werden. Natürlich kann in keinem Falle eine relevante Information erzeugt werden, die weder in der Quell-Information enthalten noch durch Normen o.ä. generell vorgegeben war.

Weitaus schwieriger ist die Lage, wenn bereits mehrere Dokumente vorliegen und Änderungen durchzuführen sind (wie es schon in der Entwicklung und dann in der Wartung laufend vorkommt). Die Funktion, die man gern hätte, aber derzeit nicht hat, wird als **„Tracing"** bezeichnet: Wenn sich der Benutzer beispielsweise entschließt, in der Spezifikation eine Änderung durchzuführen, nachdem bereits Code entwickelt wurde, sollte das Werkzeug die Anpassung des Codes möglichst gut unterstützen (im Idealfall automatische Änderung, oder wenigstens Hilfe bei der Identifikation der zu ändernden Teile).

Diese Schwierigkeit entsteht durch das Wesen der Programmentwicklung, bei der die einzelnen Dokumente (Spezifikation, Entwurf, Code und andere) nicht durch mechanische Transformation auseinander hervorgehen, sondern durch einen – bislang nur intuitiv verstandenen – Prozeß der Informationsverminderung und -anreicherung zur gleichen Zeit (siehe Bild 5).

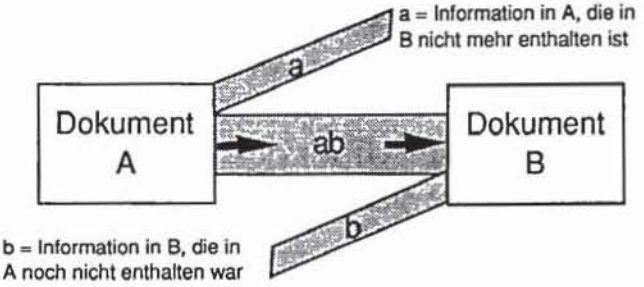

Bild 5: Die kreative Transformation

Die Abbildung zeigt schematisch, wie sich die Information von einem Dokument zum anderen verändert: Ein großer Teil (ab) bleibt erhalten, einige Information (a) fällt weg, andere (b) kommt hinzu. Wird etwa ein graphisch dargestellter Programmentwurf in Code umgesetzt, so entspricht die konkrete Graphik a, die im Entwurf nicht definierte Festlegung der Datentypen b.

Um den Übergang von A nach B vollständig zu automatisieren, muß b ohne Zutun des Entwicklers generiert werden. An einigen Stellen ist dies tatsächlich möglich: beispielsweise sind die konkreten Speicheradressen der Variablen für den Programmierer ohne Belang, sie werden durch den Compiler erzeugt. In vielen Fällen steckt aber in dieser Anreicherung (b) gerade die schöpferische Leistung des Entwicklers, sie entzieht sich natürlich der Automatisierung.

Heutige Systeme leisten das Tracing nur dann, wenn reine Generierungsschritte durchlaufen werden (d.h. wenn keine Anreicherung stattfindet oder wenn diese schematisch vorgenommen werden

kann). Die sogenannten „4th Generation Languages" beruhen auf der Idee einer reinen Generierung. Interessant wäre aber ein Ansatz für den allgemeinen Fall, doch fehlen dafür die Konzepte bisher völlig. Einzig eine (vom Benutzer kontrollierte) Verwaltung von Verweisen leistet Hilfe, doch darf man diese nur bedingt unter CASE einordnen, denn die Verweise sind nur so vollständig und sinnvoll, wie sie vom Benutzer eingegeben wurden, das Werkzeug hat keine Möglichkeit, Fehler zu erkennen und anzuzeigen.

Das Problem wird noch einmal schwieriger, wenn auch der umgekehrte Weg beschritten werden soll, wenn also nach einer Änderung am Code die früheren Dokumente angepaßt werden sollen. Solche Abläufe sind zwar absolut unerwünscht, aber in der Praxis nicht immer vermeidbar, denn gerade bei technischen Anwendungen steht oft das Entwicklungssystem (und damit das Werkzeug) nicht zur Verfügung, wenn die Software installiert oder adaptiert wird. In diesen Fällen bleibt dem Mann oder der Frau „an der Front" nicht anderes übrig, als den Code zu ändern. Es ist unwahrscheinlich, daß diese Änderungen exakt genug dokumentiert werden, um die Dokumente später konsistent zu machen.

Das *„backward tracing"* ist also eine spezielle Form des *„reverse engineering"*; es müßte Teil jedes Werkzeugsystems sein.

Der Entscheid über den Einsatz von Werkzeugen

Bei der Entscheidung für CASE sind die folgenden Vor- und Nachteile zu erwägen (am Beispiel eines Spezifikationssystems):

Probleme

- Auswahl

Werkzeuge sind nicht leicht überschaubar, und nur wenige kennen ein Werkzeug, fast niemand kennt mehrere. Wer soll also aussuchen?

- Verfügbarkeit

Oft läuft das gewählte Werkzeug nicht in der vorgesehenen Umgebung (Rechner, Betriebssystem)

- Preis

Die Systeme sind sehr teuer (erklärbar durch den hohen Entwicklungsaufwand, aber oft nicht gerechtfertigt durch die Qualität)

- Einarbeitung

Der Schulungsaufwand im weiteren Sinne (d.h. der Zeitaufwand der Mitarbeiter) ist erheblich und übersteigt in der Regel den Kaufpreis bei weitem

- Akzeptanz

Jedes Werkzeug stößt auf Widerstand, teilweise rational begründet (siehe nächsten Punkt), teilweise nicht (Programmierer sind innovationsfeindlich!)

- Qualität

Verglichen mit wirklich ausgereiften Werkzeugen (Compilern) ist die Qualität vieler CASE-Tools unbefriedigend

- Flexibilität

Jedes CASE-Tool hat seine Domäne. Gehören die Anwendungen nicht klar zu einem bestimmten Gebiet, so macht das Werkzeug Mühe

- Schnittstellen

Das Werkzeug ist mit anderen Werkzeugen in der Regel nicht in befriedigendem Maße kompatibel.

Nutzen

- Projekt- und Produktqualität

Einheitlich zeigen alle Erfahrungen positiven Einfluß auf die Qualität

- Produktivität (integral)

Die Produktivität steigt, allerdings nur, wenn man auch die späten Phasen und vor allem die Wartung berücksichtigt

- Kontrolle

Der Projektfortschritt (oder sein Fehlen) werden besser sichtbar

- Dokumentation

Es entsteht mit Werkzeug-Unterstützung bessere Dokumentation.

Die zahlreichen oben genannten Probleme wirken nicht gerade einladend für vorsichtige Leute, die sich mit dem Gedanken tragen, den Einstieg in die CASE-Welt zu wagen. Zum Trost ist aber zu sagen, daß die allermeisten Erfahrungen positiv sind und Berichte über markante Erfolge vorliegen (vgl. z.B. die Hinweise in Boehm, 1983, und in Zelkowitz et al., 1984). Als Faustregel kann man sagen: Die Wirkung von Werkzeugen ist bei weitem geringer, als die Werbung behauptet, aber bei weitem stärker, als ihre heutige Verbreitung befürchten läßt.

Ob die Einführung des Werkzeugs scheitert oder zum Erfolg führt, hängt am stärksten von zwei nicht-technischen Voraussetzungen ab:

- Es muß wenigstens *einen* Mitarbeiter geben (den **„Bannerträger"**), der selbst von dem Werkzeug begeistert ist und seinen Kollegen zeigt, wie technische Schwierigkeiten überwunden werden können. Ohne „Positive Thinking" sind heutige Werkzeuge kaum zu gebrauchen
- Projektleitung und Management müssen klar und unzweifelhaft hinter der Entscheidung zur Einführung stehen (sichere **„Rückendeckung"**).

Probleme kommen bestimmt, das Management muß dann zu seiner Entscheidung für das Werkzeug stehen, sonst ist die Übung gescheitert.

Man beachte (und bringe es dem Management rechtzeitig nahe), daß mit einem Spezifikationssystem die Entwicklung zu Beginn verzögert (!) wird, nicht beschleunigt; der Nutzen zeigt sich erst später!

Hier ist ein Zitat aus einem Aufsatz von Hatley (1983, S. 17) interessant. Er berichtet über ein Projekt, in dem Structured Analysis mit Automatendiagrammen kombiniert wurde (Hervorhebungen von mir):

Another, and possibly the most important, advantage we had was that, after the decision was made

to proceed with structured methods, there was a ***100% management committment*** *behind the effort. There would have been no time to deal with political problems which others apparently have had to contend with, and happily, none occured.*

As expected, the „up front" ***effort to prepare the requirements spec was considerably more than on previous projects****, in fact, considerably more than was originally estimated for this project. Nevertheless, the project overall is on schedule, and the results of the additional effort in terms of performance to date, and improved communication with the customer and with the design group, justify this expense.*

Die Auswahl eines Werkzeugs

Natürlich ist die Werkzeugauswahl schwierig. Kaum jemand hat Erfahrung, und schon gar nicht mit mehreren in Frage kommenden Werkzeugen. Wer aber hier versagt, richtet dreifachen Schaden an:

- Die Einführung eines letztlich unwirksamen Werkzeugs ist vergeblich, aber keineswegs umsonst, es entstehen beträchtliche Kosten, vor allem durch Schulung und Unruhe, der Kaufpreis der Systeme ist noch das wenigste
- die Einführung weiterer Werkzeuge ist blockiert, die Mitarbeiter sind für geraume Zeit nicht mehr motivierbar
- der Markt bekommt die falschen Signale, das lahmende Pferd wird getätschelt, das gute hungert.

Man sollte daher beim Vergleich verschiedener Werkzeuge nicht sparen, indem man sich auf eine billige, aber oberflächliche Evaluation durch überforderte Mitarbeiter verläßt. Eine Tabelle, in der die Prospekt-Angaben gegenübergestellt sind, ist nur ein Zeichen der Hilflosigkeit, sie lädt zum Etikettenschwindel ein.

Das Werbe-Material ist bei der Auswahl leider alles andere als hilfreich. Ich werde regelmäßig von Agressionen heimgesucht, wenn ich mich durch einen Hochglanzprospekt gequält habe, ohne eine einzige handfeste Information zu finden. Als Ingenieur setze ich meine Hoffnungen vor allem auf schematische Darstellungen, aber ich werde auch von diesen meist frustriert: In der Epoche der Postmoderne mußten wir uns daran gewöhnen, Fassaden zu sehen, die Selbstzweck sind, nicht das Gesicht des Hauses, sondern seine Maske; auf unserem Gebiet entsprechen dem die entfesselten Graphiken, die zwar stilsicher komponiert und mit 3D-Effekt versehen sind, aber leider keine sinnvolle Semantik besitzen. Offenbar ist auch hier das Medium bereits die Botschaft, und zwar oft die einzige.

Heute wünschen viele Kunden „objektorientierte Werkzeuge", auch wenn sie wohl nicht immer genau wissen, was das sein sollte; die Anbieter übernehmen folgerichtig dieses Wort als schmückendes Attribut (Frei nach dem alten Ingenieurspruch: *„Vor sechs Wochen wußte ich noch nicht, wie man ‚Opjeckt-orientiert' schreibt, und nun biete ich es schon an"*).

Um zu einer seriösen Bewertung zu kommen, muß man die Werkzeuge von ihrer Architektur her analysieren und gegen die eigenen Anforderungen prüfen; diese können erst *nach* Wahl der einzusetzenden Methoden und Sprachen formuliert werden. Wo die erforderliche Kompetenz dafür nicht im Hause ist, sollte man lieber externe Hilfe in Anspruch nehmen.

Natürlich muß die Bewertung am Ende auch in die Praxis umgesetzt werden (vgl. N.N., 1991).

Ausblick

Das traditionelle Dilemma des Software Engineerings liegt darin, daß der Weg des geringsten Widerstands *(„Verschiebe nichts auf morgen, was Du auch auf übermorgen verschieben könntest!")* der Holzweg der Software-Entwicklung ist (fehlende Planung, fehlende Dokumentation, fehlende Kontrolle). Ein Werkzeug sollte den Effekt haben, daß dieser Widerspruch vermindert wird, weil *mit* dem Werkzeug das *richtige* Verhalten auch *einfach* wird.

Der zukünftige Erfolg der Werkzeuge wird wesentlich davon abhängen, wie weit die oben skizzierten Probleme (Speicherung, Tracing) gelöst werden können. Solange das Defizit auf dem Gebiet der Metriken bestehen bleibt, werden sich die Werkzeuge nur dort verbreiten, wo ein ausreichend initiatives Management die notwendigen Rahmenbedingungen schafft. Firmen, denen es gelingt, *parallel* Fortschritte auf den Gebieten der Standardisierung, der Qualitätssicherung, der Entwicklungsmethodik und des Werkzeugeinsatzes zu erzielen, werden einen erheblichen Wettbewerbsvorteil haben.

Literatur

Nachfolgend sind die zitierten Arbeiten aufgelistet. Zum Thema CASE gibt es kaum spezielle Literatur (vgl. Fisher, 1988; Chikofsky, 1988; Österle, 1988; Balzert (1990). Über Software Engineering gibt es einige Bücher, z.B. von Fairley (1989) und Sommerville (1985). Die sehr teuren Übersichten von Ovum (Rock-Evans, 1989) kenne ich nur aus der Werbung.

Abramowicz, K., K. Dittrich, W. Gotthard, M. Härtig, R. Langle, M. Ranft, T. Raupp, S. Rehm (1988): DAMOKLES (Database Management System for Design Applications). Reference Manual, Release 2.0, FZI Karlsruhe

Balzert, H. (1990): CASE – Systeme und Werkzeuge. Bibliographisches Institut, Mannheim

Boehm, B. W. (1983): Seven basic principles of Software Engineering. Journal of Systems and Software, 3, 3–24

Cameron, J. R. (1983): JSP & JSD: The Jackson Approach. IEEE Tutorial, IEEE Comp. Soc. Press, Order No. 516

Chikofsky, E. J. (1988): Computer-Aided Software Engineering (CASE). IEEE Computer Society, 13, Av. de l'Aquilon, B-1200 Brüssel, Order No. FX1917

Dart, S. A., R. J. Ellison, P. H. Feiler, A. N. Habermann (1987): Software Development Environments. IEEE COMPUTER, November 1987, pp. 18–28, speziell 22–23

deMarco, T. (1978): Structured Analysis and System Specification. Yourdon Press, New York

Fairley, R. (1985): Software Engineering Concepts. McGraw-Hill Book Company, New York

Fisher, A. S. (1988): CASE: Using Software Development Tools. John Wiley & Sons, New York

Fruhauf, K., J. Ludewig, H. Sandmayr (1991): Software-Projektmanagement und -Qualitätssicherung. vdf, Zürich, und Teubner, Stuttgart, 2. Aufl.

Hatley, D. J. (1983): A structured analysis method for large, real-time systems. Lear Siegler Inc., Grand Rapids, Michigan (unveroffentlicht)

Howden, W. (1982): Contemporary software development environments Comm. ACM, 25, 5, 318–329

Hünke, H. (ed.) (1981): Software Engineering environments. North Holland Publishing Company, Amsterdam, New York, Oxford

IEEE (1983): Standard glossary of software engineering terminology. IEEE Std 729-1983

Jackson, M. A. (1975): Principles of program design. Academic Press, London, New York, San Francisco. auf Deutsch: Jackson, M. A. (1979): Grundsätze des Programmentwurfs. S. Toeche-Mittler Verlag, Darmstadt

Ludewig, J. (1982): Computer aided specification of process control software. IEEE COMPUTER, Mai 1982, 12–20

Matheis, H. (1990): Informationsverwaltung für Software-Projekte. Dissertation, Universität Stuttgart

McMenamin, S. M., J. F. Palmer (1988): Strukturierte Systemanalyse. Hanser und Prentice Hall International, London. (Originalausgabe 1984 bei Yourdon, New York)

N.N. (1991): Auswahl von Methoden und Werkzeugen für Projekte der innerbetrieblichen Informationsverarbeitung. Beitrag 11 in J. Ludewig (Hrsg.): Software- und Automatisierungsprojekte – Beispiele aus der Praxis. Teubner, Stuttgart, 205–218.

Österle, H. (Hrsg.) (1988): Anleitung zu einer praxisorientierten Software-Entwicklungsumgebung. Bande 1 und 2. Angew. Informationstechnik Verlags GmbH, Halbergmoos

Riddle, W. (1985): Software Environments. Tutorial im Rahmen der 8th ICSE, London

Rock-Evans, R. (1989): CASE Analyst Workbenches: a detailed product evaluation. Ovum Ltd, 7 Rathbone Street, London W1P 1AF, England. (550 £. In gleicher Art bei Ovum: CASE: Commercial Strategies, fur 385 £)

Ross, D. T. (1985): Applications and extensions of SADT. IEEE COMPUTER 18, 4, 25–34

Sommerville, I. (1989): Software Engineering. Addison-Wesley Publ. Co., London usw., 3rd ed.

Teichroew, D., E. A. Hershey III (1977): PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems. IEEE Trans. Software Eng., SE-3, 41–47

Zelkowitz, M. V., R. T. Yeh, R. G. Hamlet, J. D. Gannon, V. R. Basili (1984): Software Engineering Practices in the US and Japan. IEEE COMPUTER, 1984, June, 57–66