Gales

Defizite im Software Engineering

S.Wendt
Universität Kaiserslautern

1. Zur Motivation

In der Eröffnungsveranstaltung der 22. GI–Jahrestagung in Karlsruhe meinte der GI–Präsident, daß die in den letzten Jahrzehnten erreichten Fortschritte auf dem Gebiet des Software Engineering ähnlich spektakulär seien wie die Fortschritte in der Luft– und Raumfahrt, die von Lilientals Flugmaschine zum Space Shuttle führten. Angesichts dieses Urteils eines prominenten Informatikers mußte sich ein Autor, der schwerwiegende Defizite im Software Engineering festgestellt zu haben glaubt, natürlich fragen, ob es nur an ihm selbst liege, daß er die Lage so völlig anders beurteilt. Er konnte diese Frage aber verneinen, da er eine beträchtliche Anzahl von Personen in leitenden Positionen in der industriellen Software–Entwicklung kennt, die mit ihm bezüglich der Einschätzung der Misere völlig übereinstimmen. Vielleicht kann der vorliegende Aufsatz ein wenig helfen zu verstehen, wieso es zu derart gegensätzlichen Beurteilungen ein und derselben Lage kommen kann.

2. Engineering – ein erklärungsbedürftiger Begriff

Die Entwicklung von Software ist ein schöpferischer Akt. Mit der Bildung des Begriffs Software Engineering wurde zum Ausdruck gebracht, daß der schöpferische Akt der Softwareentwicklung mehr der Tätigkeit von Ingenieuren und weniger der Tätigkeit von Künstlern oder Handwerkern entsprechen solle. Die Einsicht, daß die Probleme der Softwareentwicklung nur beherrscht werden können, wenn die Software "ingenieurmäßig" entwickelt wird, ist schon über 20 Jahre alt. Aber was heißt denn ingenieurmäßig?

Zur Beantwortung dieser Frage muß man analysieren, was Ingenieure tun. Dabei stellt man fest, daß die Ingenieurtätigkeiten in sehr unterschiedliche Klassen einzuteilen sind. Für die vorliegenden Betrachtungen genügt es, zwischen mathematisch orientierten Tätigkeiten und anderen Tätigkeiten zu unterscheiden. Typische Beispiele aus diesen beiden Tätigkeitsklassen sind die Baustatik auf der einen Seite und der Grundrißentwurf oder die Baustellenorganisation auf der anderen Seite.

Es ist nicht üblich, für alles, was Ingenieure tun, im Deutschen das Wort Engineering zu verwenden, obwohl dieses Wort seit vielen Jahren als Fremdwort im Deutschen akzeptiert ist. Es wird üblicherweise nur mit einschränkenden Zusätzen wie in Systems Engineering oder Software Engineering gebraucht. Nichtingenieure assoziieren zum Wort Engineering oft nur die Anwendung mathematischer Verfahren zum Zwecke der Gestaltung technischer Produkte, denn dies sehen sie als das typische Merkmal der Ingenieurstätigkeit an. Andererseits bezeichnet das Fremdwort Engineering im Sprachgebrauch deutschsprachiger Ingenieure häufig speziell solche Tätigkeitsbereiche, in denen mathematische Verfahren keine oder nur eine geringe Rolle spielen. Es besteht somit die Gefahr von Mißverständnissen, und deshalb ist es zweckmäßig, vorab zu definieren, in welchem Sinne man das Wort Engineering gebrauchen wird. Eine Definition, die man den folgenden Betrachtungen zugrundelegen kann, findet man in [DEN 91]:

> *Software Engineering* ist die Anwendung wissenschaftlicher Erkenntnisse mit dem Ziel, Computer mittels Programmen, Verfahren und zugehörigen Dokumenten dem Menschen nutzbar zu machen.

3. Komplexitätsbeherrschung als das zentrale Problem

Es ist die Kernaussage dieses Aufsatzes, daß die Softwaresysteme im oberen Bereich der Komplexitätsskala heute noch nicht ingenieurmäßig beherrscht werden.

Wenn der Autor hier von komplexen Softwaresystemen spricht, so hat er dabei Systeme vor Augen, deren Entwicklung über 100 Mannjahre in Anspruch genommen hat und deren Code–Umfang mehr als eine halbe Million Zeilen umfaßt. Wenn man einen solchen Code–Umfang ausdruckt, ergibt sich ein Papierstapel von über einem Meter Höhe. Der Autor kennt einige derartige industrielle Systeme recht gut, weil er an ihrer Dokumentation beteiligt war bzw. noch ist.

Aus der Tatsache, daß derart komplexe Software realisiert und vermarktet wird, darf man selbstverständlich nicht schließen, daß sie dann auch zwangsläufig ingenieurmäßig beherrscht werde. Die ingenieurmäßige Beherrschung äußert sich nämlich nicht daran, was der Benutzer sieht. Der Benutzer mag durchaus zu akzeptablen Preisen ein System mit befriedigender Funktionalität und Robustheit bekommen, und er mag auch erleben, daß die Systempflege für ihn keine schwerwiegenden Probleme mit sich bringt. Die Frage nach der ingenieurmäßigen Beherrschung wird erst durch den Blick hinter die Kulissen entschieden.

Dabei wird man feststellen, daß nur diejenigen Probleme ingenieurmäßig beherrscht werden, die den Komponenten zuzuordnen sind. Diejenigen Probleme aber, die gar nicht existieren würden, wenn das System nicht so komplex wäre, werden nur irgendwie – mehr schlecht als recht – entschärft, aber ingenieurmäßig beherrscht werden sie nicht – zumindest hinter keiner der vielen Kulissen, hinter die der Autor hat blicken können. Alle diese Probleme, deren Existenz eine Folge der Komplexität ist, haben einen gemeinsamen Kern:

> Der Erwerb des für eine Mitwirkung an der Systementwicklung oder –pflege erforderlichen Präsenzwissens ist extrem ineffizient, weil der Informationsbedürftige viel zu sehr auf mündliche Überlieferung angewiesen ist.

Dies äußert sich insbesondere in der Vormachtstellung von "Gurus", die aufgrund der Tatsache, daß sie vom Anfang der Entwicklung an dabei waren, einen Wissensvorsprung haben, den die später Hinzugekommenen nicht mehr aufholen können. Es gehört aber zum Wesen ingenieurmäßiger Verfahrensweisen, daß der einzelne Ingenieur – im Gegensatz zum Künstler – grundsätzlich austauschbar bleibt, d.h. daß verfahrensmäßig verhindert wird, daß Informationsmonopole bei einzelnen Personen entstehen.

In [BRO 75] findet man die Behauptung: "Adding manpower to a late software project makes it later!" Als Beschreibung einer zeitbedingten Situation kann diese Aussage zweifellos zutreffen, aber als zeitunabhängiges Gesetz braucht man sie nicht hinzunehmen. Denn sonst hätte man ja resignierend akzeptiert, daß ein wesentliches Ziel des Software Engineering nicht erreichbar sei.

4. Überbewertung des Formalisierbaren

Im vorstehenden Abschnitt wurde gesagt, daß die ingenieurmäßige Beherrschung auf der Komponentenebene heute gegeben sei. Dies ist die Konsequenz der beeindruckenden Fortschritte auf den Gebieten der Algorithmen, der Sprachen und der formalen Verfahren. Nun darf man aber diesen Fortschrittsverlauf nicht einfach in den Bereich der komplexen Systeme extrapolieren, d.h. man darf nicht erwarten, weitere Fortschritte auf den genannten Gebieten würden auch zur ingenieurmäßigen Beherrschung derjenigen Probleme führen, die im riesigen Code–Umfang der komplexeren Softwaresysteme begründet sind. Denn um Algorithmen und formale Verfahren ins Spiel bringen zu können, muß man Elemente identifizieren können, die meistens erst auf der Komponentenebene definiert sind. Durch eine Analogie veranschaulicht heißt dies, daß bei der Projektierung eines Fernsehsendezentrums abgesehen von Grobkalkulationen sehr wenig gerechnet wird, solange es um das Ganze und nicht um die Komponenten geht. Mächtige mathematische Verfahren kommen erst zum Einsatz, wenn die Filter, Verstärker, Antennen und ähnliche Komponenten entwickelt werden müssen. Auch bei komplexeren Bauwerken können die Baustatiker erst zu rechnen beginnen, nachdem eine Systemzerlegung in Betonflächen, T–Träger und ähnliche Komponenten vorgenommen wurde.

Ein komplexes Softwaresystem besteht meist aus Tausenden von Komponenten, und in den einzelnen Komponenten findet man die formulierten Algorithmen. In diesen Systemen gibt es keinen "Algorithmus des Systems", von dem aus man durch schrittweise Verfeinerung zu den Algorithmen der Komponenten gelangen würde. Zwar kommt die schrittweise Verfeinerung von Algorithmen in diesen Systemen dauernd vor, aber in diesen Fällen ist der Algorithmus, von dem man ausgeht, immer schon der Algorithmus einer Komponente.

Man darf realistischerweise nicht annehmen, ein wirklich komplexes System, dessen Komplexitätsbeherrschung für die Software–Ingenieure ein besonderes Problem darstellt, könne vor seiner Implementierung formal spezifiziert werden.

Der Vollständigkeit wegen sei angemerkt, daß hier selbstverständlich nicht diejenigen Fälle interessieren, bei denen die Spezifikationsfreiheit so stark eingeschränkt ist, daß der Spezifizierende nur noch eine vorgegebene Liste von Fragen abarbeiten muß, deren Beantwortung keine große Mühe macht. In diesen Fällen geht es nämlich nur noch um die Auswahl eines Systems aus einer vordefinierten Systemklasse. Man denke an Software für betriebswirtschaftliche Anwendungen, wo der Spezifizierende nur noch Bildschirmmasken, Dialogschrittfolgen und Datenbankschemata festlegt. Im Rahmen des vorliegenden Aufsatzes wird dagegen mit dem Begriff Spezifikation immer die Vorstellung verbunden, daß der Spezifizierende Beliebiges verlangen kann, was mit Informatikmitteln realisierbar ist.

Weder ein Jumbojet noch ein Großkraftwerk werden vor ihrer Konstruktion in einer solchen Form beschrieben, daß daraus mit formalen Verfahren irgendwelche interessante Folgerungen abgeleitet werden könnten. Selbstverständlich wird es in diesen Fällen eine lange Anforderungsliste geben, worin unter anderem die Anzahl der Sitzplätze bzw. die zu liefernde elektrische Leistung vorgeschrieben wird, aber formal ableiten läßt sich daraus gar nichts.

Um sich vor Augen zu führen, daß der Versuch einer vollständigen formalen Spezifikation eines komplexen Systems aus praktischen Gründen scheitern muß, braucht man nur die folgenden Überlegungen anzustellen:

Ein System, das von 50 Entwicklern in einem Zeitraum von 4 Jahren entwickelt wurde, umfaßt rund eine Million Zeilen Quellcode in der Sprache C, die sich auf 8.000 Funktionsprozeduren

aufteilen. Obwohl es nicht geschah, hätte zu jeder Funktion, bevor sie codiert wurde, eine formale Spezifikation geschrieben werden können. Vielleicht hätten im Mittel 15 Zeilen Spezifikation pro Funktion ausgereicht, so daß sich insgesamt 120.000 Zeilen Spezifikation ergeben hätten. Darin wäre sicher etliches enthalten, was nur implementierungsspezifisch ist und in einer Systemspezifikation nicht vorkäme. Man kann zwar eine Systemspezifikation nicht durch bloße Auswahl aus diesen 120.000 Zeilen Funktionsspezifikation gewinnen, aber man kann daraus doch, indem man einen implementierungspezifischen Anteil annimmt, den Umfang der Systemspezifikation abschätzen. Wenn man beispielsweise mit einem unwahrscheinlich hohen implementierungsspezifischen Anteil von 80% rechnet, ergibt sich ein Umfang der Systemspezifikation von 24.000 Zeilen.

Selbstverständlich dürfen diese Spezifikationszeilen nicht mit den oben betrachteten Implementierungszeilen bezüglich des Erstellungsaufwands gleichbehandelt werden. Beim Erstellen einer Vorabspezifikation des gesamten Systems bestünde die unbewältigbare Schwierigkeit darin, daß man eine gewaltige Fülle von unbewußt Impliziertem explizit machen müßte. Diese Schwierigkeit gibt es beim Erstellen der 120.000 Zeilen Funktionsspezifikation nicht, denn wenn man einmal entwerfend bis zur einzelnen Funktion vorgestoßen ist, dann hat man bereits sehr viel ursprünglich Impliziertes explizit gemacht.

Das inkrementelle Spezifizieren im Laufe des Entwerfens und Implementierens ist selbstverständliche Praxis in vielen Auftraggeber/Auftragnehmer–Beziehungen, und dies wäre nicht so, wenn sich diese Vorgehensweise nicht bewährt hätte. Man betrachte nur das einfache Beispiel der Planung und Erstellung eines Einfamilienhauses. Die Entscheidung, wo der Bauherr die elektrischen Steckdosen plaziert haben möchte, fällt ihm am leichtesten, wenn der Rohbau bereits steht, denn dann kann er die gewünschten Positionen mit Kreide auf die Wände zeichnen. Wenn der Bauher alles, worauf er Einfluß nehmen will – und genau dies müßte der Inhalt der Spezifikation sein – vor dem ersten Spatenstich aufschreiben müßte, dann würde er entweder nie fertig, weil er immer weitergrübeln würde, ob er denn auch nichts vergessen habe, oder er würde seine Spezifikation zwar abliefern, aber hinterher feststellen müssen, daß er Wesentliches impliziert, aber nicht aufgeschrieben hat.

Die Tatsache, daß die Softwareentwicklung selbstverständlich in einer computerbasierten Entwicklungsumgebung geschieht, führt leicht dazu, daß die Software–Ingenieure meinen, ihre Probleme nur noch derart lösen zu können, daß dabei der Computer die zentrale Rolle spielt. Es wäre ja auch wunderschön, wenn man geeignete Programmiersprachen einerseits und Programme zur Steuerung des Entwicklungsprozesses andererseits hätte, die beide gemeinsam dafür sorgen würden, daß das, was die Entwickler an ihren Bildschirmarbeitsplätzen eingeben, nicht nur das gewünschte Programmsystem ist, sondern auch noch für die automatische Generierung sämtlicher interessierenden Dokumente ausreicht. Da aber der Computer nicht feststellen kann, ob die anschauungssemantischen Erfordernisse befriedigt sind, sollte man sich bei der Lösungssuche nicht derart werkzeugorientiert einschränken. Denn bei dieser Einschränkung sieht man die Menschen zwangsläufig nur noch als Kommunikationspartner des Computers, so daß die Kommunikation der Menschen untereinander aus dem Blickfeld gerät. Die aktuellen Probleme des Software–Engineering sind aber vorwiegend Kommunikationsprobleme der Menschen untereinander.

5. Unterbewertung der Anschauung

Es wurde schon gesagt, daß es bei der Komplexitätsbeherrschung letztlich um die Frage nach dem Präsenzwissen geht, welches die Beteiligten über das System besitzen müssen, und nach dem Weg, auf dem dieses Präsenzwissen erworben wird. Komplexitätsbeherrschung ist garantiert nicht gegeben, wenn alle relevante Information nur wie in einem Lexikon gespeichert ist. Denn wer zu wenig oder ungeeignetes Präsenzwissen hat, der wird in einer Erklärung zu einem unbekannten Begriff weitere unbekannte Begriffe finden, die er nachschlagen muß, und dieser Lawineneffekt läßt keinen inkrementellen Wissenszuwachs und auch keine vergröberten ganzheitlichen Sichten entstehen.

Die Möglichkeit, sämtliche relevanten Informationen im Computer online für die Anzeige am Bildschirm verfügbar zu haben, hat zwar große Vorteile, so daß man darauf nicht verzichten kann; man sollte sich aber auch einer Gefahr bewußt sein, die damit verbunden ist. Diese Gefahr besteht in der Suggestion, daß ein Präsenzwissen über die Strukturzusammenhänge überflüssig sei. Man denke an Hypertext als eine Menge kleiner Textabschnitte, wo im Extremfall jedem Wort in einem Textabschnitt wieder ein oder mehrere Textabschnitte zugeordnet sind, die man durch einfaches Anklicken des Wortes auf den Bildschirm holen kann. Auf diese Weise bewegt man sich durch eine Welt, deren Landkarte man nicht kennt und die man beim Wandern auch nicht erfassen kann. Wenn einem also an der Kenntnis der Landkarte gelegen ist – und im Interesse der Komplexitätsbeherrschung muß einem daran gelegen sein –, dann muß man sich um die Gestaltung der Landkarte und deren Verbreitung bewußt bemühen.

Wenn man nach der Beschaffenheit von Präsenzwissen und nach dem Prozeß des Wissenserwerbs fragt, dann erkennt man sehr bald die zentrale Rolle, welche die Anschauung dabei spielt. Dies ist eine Konsequenz der Tatsache, daß unsere Augen den bei weitem mächtigsten Informationskanal bilden, der uns mit unserer Umgebung verbindet. Deshalb gelingt uns die Erfassung von Strukturen immer nur dann besonders gut, wenn wir eine Anschauung damit verbinden können. Weshalb würde man sonst beispielsweise einen Stammbaum zeichnen, wo doch Abstammung ursprünglich keine räumliche Relation ist? Dort, wo eine primäre Anschauung fehlt – und das ist bei allen Strukturen in informationellen Systemen der Fall –, muß man eine sekundäre Anschauung herbeiführen, wenn es sich um einen für das Präsenzwissen relevanten Bereich handelt.

Das bedeutet nicht, daß man für jede Prozedur ein Ablaufdiagramm zeichnen muß, denn da sich ohnehin bei jedem, der den Quellcode der Prozedur liest, implizit eine Anschauung damit verbindet, erübrigt sich eine explizite Graphik. Explizite Graphik ist nur dann erforderlich, wenn es um Strukturen geht, zu denen ein Quellcodeumfang gehört, den man beim Lesen nicht mehr überschauen kann, denn dann kann implizit keine Anschauung mehr entstehen.

Die Vorstellung, die benötigten graphischen Darstellungen automatisch aus dem Quellcode generieren zu können, ist unrealistisch, denn dabei werden die besonderen Anschauungsbedürfnisse des Menschen unterschätzt. Bei der automatischen Layoutgestaltung können keinerlei anschauungssemantische Bezüge berücksichtigt werden; für die ganzheitliche Strukturerfassung durch den Menschen ist es aber von wesentlicher Bedeutung, daß sich solche Bezüge in räumlichen Nachbarschaften, Fluchtlinien oder anderen Layoutmerkmalen äußern. Daß die Bedeutung der Layoutgestaltung oft unterschätzt wird, kann man auch beim Studium mancher Informatikbücher erkennen, in denen die darin verwendeten Strukturgraphiken häufig so unstrukturiert gestaltet sind, daß sie deutlich das Desinteresse des Autors an der Graphik dokumentieren.

Bei der Behandlung derjenigen Probleme des Software Engineering, die nur unter Einbeziehung von Anschauungssemantik gelöst werden können, kann der Computer keine große Hilfe sein, denn diesem ist Anschauung so fremd wie dem Blinden die Farbe.

Wenn ein Mensch eine sprachliche Form interpretiert, dann ordnet er der wahrgenommenen Form eine Anschauung zu; für ihn gilt also eine Semantik, die hier Anschauungssemantik genannt wird. Für den Computer gilt dagegen eine formale Semantik, d.h. er ordnet der zu interpretierenden Form als Interpretationsergebnis wieder eine Form zu. Für den Computer ist es irrelevant, ob anstelle der anschaulichen Bezeichnungen PUSH und POP für die damit gemeinten Stackoperationen die Zeichenfolgen KLAF und GUR verwendet werden; der Mensch dagegen braucht die anschaulichen Bezeichnungen, weil er den Stack als endliche Menge aufeinanderliegender Elemente vor Augen haben will.

Da der Computer keine Anschauungssemantik kennt, kann er auch nicht dazu benutzt werden sicherzustellen, daß bei der Wahl von Namen für Module, Funktionen, Typen oder Variable zweckmäßige anschauungssemantische Bezüge hergestellt werden. Je komplexer die Systeme werden, umso undurchschaubarer wird der Namenswirrwarr, wenn den einzelnen Entwicklern die Namenswahl freigestellt bleibt, d.h. wenn die Sicherstellung anschauungssemantischer Bezüge nicht bewußt als Engineering–Aufgabe wahrgenommen wird. Man bedenke, daß in der weiter oben erwähnten Software in einer Million Zeilen C–Quellcode insgesamt rund 29.000 Namen vereinbart werden mußten.

Nicht nur für das Verständnis bei Lesen von Quellcode sind anschauungssemantische Bezüge erforderlich, sondern insbesondere auch für das Vergröbern von Systemdarstellungen. Es gehört zum Wesen komplexer Systeme, daß ein Mensch den Zusammenhang zwischen dem Systemverhalten und dem Systemaufbau aus Komponenten mit bekanntem Verhalten nicht mehr als Ganzes erfassen kann. Derartige Systeme könnten gar nicht zielstrebig konstruiert und im Betrieb beherrscht werden, wenn es dem Menschen nicht möglich wäre, sich vergröberte Systemdarstellungen zu schaffen, die er wieder als Ganzes erfassen kann. Bei solchen Vergröberungen geht es darum, alle Details wegzulassen, die für ein intuitives Grobverständnis nicht gebraucht werden.

Eine *Vergröberung* ist etwas anderes als eine *Abstraktion*. Alles was in einer abstrahierten Systemdarstellung ausgesagt wird, trifft ohne Abstriche auf das konkrete System zu. So könnte beispielsweise in einer abstrahierten Systemdarstellung von einer Komponente mit der Zuordnerfunktion $y = f(x)$ die Rede sein, wobei im konkreten System $f(x) = 5x + 1$ gilt. Dagegen müssen Aussagen, die in einer vergröberten Systemdarstellung gemacht werden, nicht unbedingt auf das konkrete System exakt zutreffen. So könnte beispielsweise auch in einer vergröberten Systemdarstellung von einer Komponente mit der Zuordnerfunktion $y = f_{GROB}(x)$ die Rede sein, wobei aber im konkreten System neben der Variablen x noch eine zweite Variable z im Argument der Funktion $f_{PRÄZIS}(x,z) = 5x+z$ vorkommt. Beim Übergang von einer abstrahierten Systemdarstellung zum konkreten System muß man detaillieren, also weitere Aussagen hinzubringen, ohne bereits gemachte Aussagen zu revidieren, wogegegen man beim Übergang von einer vergröberten Systemdarstellung zum konkreten System präzisieren muß, was bedeutet, daß man bereits gemachte Aussagen teilweise wieder revidiert.

Ohne intensive Verwendung vergröberter Systemdarstellungen sind komplexe Systeme nicht beherrschbar. Die Entscheidung, wie zweckmäßigerweise vergröbert werden soll, d.h. an welchen Stellen man auf Präzision zugunsten von ganzheitlicher Erfaßbarkeit verzichtet, entzieht

sich verständlicherweise jedem Formalismus. Es gibt also viele dringend benötigte Systemdarstellungen, die nicht automatisch aus dem Quellcode generiert werden können.

Der Bedarf an anschaulichen Strukturplänen fällt unmittelbar ins Auge, wenn man durch die Büros einer Softwareentwicklungsabteilung geht: Die Tafeln, die dort an den Wänden hängen, sind meistens dicht mit Texten, Tabellen und Zeichnungen belegt. Den Tafelinhalten kann man mühelos entnehmen, daß man sich in einer Softwareabteilung befindet, so wie man andernorts anhand von Tafelinhalten sicher darauf schließen kann, sich in einer Entwicklungsabteilung der Elektrotechnik oder des Maschinenbaus zu befinden. Während aber die Strukturpläne der Elektrotechnik und des Maschinenbaus nicht nur auf den Tafeln erscheinen, sondern selbstverständliche Objekte der Engineering–Verfahren sind, werden die Zeichnungen auf den Tafeln der Softwareabteilungen nur als flüchtige Skizzen in den Diskussionen benützt und spielen als bewußt eingeplante Engineering–Objekte – abgesehen von wenigen Sonderfällen – praktisch keine Rolle. Man kann die Problematik zu dem Satz verdichten: "Sagt mir, anhand welcher Dokumente in Eueren Entwicklungsbesprechungen diskutiert wird, und ich sage Euch, wie es mit Euerem Engineering steht!"

Eine ernstgenommene Disziplin hat das Bemühen um Veranschaulichung bisher noch nicht werden können. Zum einen gibt es auf diesem Gebiet zu viel Dilettantisches in Form sogenannter Systemarchitekturbilder, die nur aus neben–, über– und ineinander gesetzten Quadern bestehen. Zum anderen sind immer noch zuviele maßgebliche Informatiker überzeugt, daß nur dasjenige eine ernstzunehmende Disziplin sein könne, was auf einer mathematischen Basis ruht, und daß deshalb nur die alphanumerischen Zeichenketten als ernstzunehmende Darstellungen in Frage kämen. So findet man beispielsweise in [HOA 86] die Aussage: "Computer programs are mathematical expressions. ... Such pictures actually inhibit the use of mathematics in programming, and I do not approve of them. ... excessive reliance on pictures ... would not be regarded as a good qualification ... It is equally inappropriate for a professional programmer." Und in [BRO 86] wird behauptet: "Software is invisible and unvisualizable." Solange die in diesen Zitaten zum Ausdruck kommenden Überzeugungen nicht wirklich überwunden sind, kann es nicht zur Selbstverständlichkeit werden, an den Wänden von Softwareentwicklungsabteilungen aussagekräftige Strukturpläne hängen zu sehen. Solange dies aber nicht der Fall ist, besteht ein schwerwiegendes Defizit im Software Engineering.

6. Unterschätzung verfahrensorientierter Arbeitsteilung

Da an der Entwicklung eines komplexen Softwaresystems selbstverständlich immer viele Menschen beteiligt sind, stellt sich hier nicht die Frage, ob Arbeitsteilung sinnvoll ist, sondern nur, ob sie angemessen organisiert ist. Zur Beantwortung dieser Frage vergleicht man zweckmäßigerweise die Situation beim Software Engineering mit der Situation beim Engineering in den traditionellen technischen Bereichen. Man denke an die Entwicklung eines Jumbojet, die von vielen Ingenieuren gemeinsam durchgeführt wird.

Für diesen Vergleich ist es hilfreich, zwischen *komponentenorientierter* und *verfahrensorientierter Arbeitsteilung* zu unterscheiden. Komponentenorientierte Arbeitsteilung liegt vor, wenn zwei Menschen mit gleichem Verfahren an zwei unterschiedlichen Systemkomponenten arbeiten – wenn also beispielsweise der eine Konstruktur die Triebwerksaufhängung und der andere die Fahrwerkskammer konstruiert bzw. wenn der eine Programmierer die Speicherverwaltung und der andere die Dialogsteuerung entwickelt. Verfahrensorientierte Arbeitsteilung liegt dage-

gen vor, wenn zwei Menschen für die Abwicklung unterschiedlicher Verfahren zuständig sind – wenn also beispielsweise der eine Ingenieur konstruierend und der andere qualitätsprüfend tätig ist bzw. wenn der eine Informatiker Komponenten entwickelt und der andere Systeme integriert.

Viele Software–Ingenieure lassen sich von der Ansicht irreleiten, daß man verfahrensorientierte Arbeitsteilung ohne Schaden "virtualisieren" könne. Was dies bedeutet, versteht man am ehesten anhand eines einfachen Beispiels: Man betrachte die Arbeitsteilung zwischen einem Architekten und einem Maurer. Zwischen beiden besteht ein institutionalisierter Informationsfluß. Wenn nun ein und derselbe Mensch abends ein Häuschen mauert, das er tagsüber geplant hat, dann wurde die Arbeitsteilung zwischen dem Architekten und dem Maurer virtualisiert. Dabei ging die Selbstverständlichkeit des institutionalisierten Informationsflusses verloren. Während der Architekt bei der realen Arbeitsteilung motiviert ist, exakte Pläne zu erstellen, weil sie der Maurer braucht, geht diese Motivation verloren, wenn der Planer anschließend selbst mauert. Deshalb ist es auch unrealistisch, von einem Programmierer die Motivation zu erwarten, eine Funktionsprozedur, die er implementieren muß, vorher oder nachher exakt zu spezifizieren.

Denn virtuelle Arbeitsteilung mit dem Zwang, an den Phasengrenzen Dokumente bestimmter Art – Pflichtenheft, Design–Spezifikation, Test–Anweisung u.a.m. – abzuliefern, bringt die Entwickler dauernd in unerträgliche innere Konfliktsituationen. Der Entwickler, der jetzt Designer sein soll, aber auch schon weiß, daß er anschließend Codierer sein wird, kommt sich wie in einer Zwangsjacke vor, wenn ihn das Management oder die Projektlenkungswerkzeuge zwingen, auf Erfolgserlebnisse noch lange zu warten, die er als Codierer jetzt schon haben könnte. Denn Erfolgserlebnisse ergeben sich vorwiegend aus den Endergebnissen und nicht aus den Zwischenergebnissen der eigenen Arbeit. Deshalb bezieht derjenige, der durch seine eigene Arbeit eine Softwarekomponente letztendlich lauffähig macht, sein Erfolgserlebnis unmittelbar aus dem laufenden Programm. Daher kann die sorgfältige Erstellung anschaulicher Dokumente nicht zur allgemein akzeptierten Selbstverständlichkeit werden, solange es keine reale verfahrensorientierte Arbeitsteilung gibt, bei der bestimmte Mitarbeiter verfahrensbedingt ihre Erfolgserlebnisse nur aus der Qualität der von ihnen geschaffenen Dokumente beziehen können.

Zu der Frage, welche Form der verfahrensorientierten Arbeitsteilung in der Softwareproduktion in Abhängigkeit von der Komplexität des Produkts gewählt werden soll, gibt es noch keine ausreichend fundierten Antworten. Die Maschinenbauer und die Elektrotechniker hatten über ein Jahrhundert Zeit, Ingenieurverfahren auszudenken und in der Praxis zu erproben; die Informatiker hatten diese Zeit noch nicht. Wegen der Besonderheiten des Produkts Software war es nicht möglich, einfach die bewährten Verfahren aus den anderen technischen Bereichen zu übernehmen. Deshalb braucht man zwangsläufig Zeit zum Experimentieren. Denn ob ein Verfahren zweckmäßig ist oder nicht, kann sich erst bei der Anwendung in der Praxis zeigen. Man kann zwar leicht an der Universität oder in Planungsstäben von Firmen Ideen entwickeln, wie man die Arbeitsteilung gestalten solle, aber es ist äußerst aufwendig und risikoreich, diese Ideen einer praktischen Bewährungsprobe zu unterwerfen. Das ist nur machbar, wenn die Ideen so überzeugend sind, daß alle an einem Großprojekt der Softwarentwicklung Beteiligten einen Vorteil darin sehen, gemäß dieser Ideen zu verfahren. Denn nicht nur die Manager, sondern auch die Entwickler müssen positiv mitwirken, weil bereits der passive Widerstand von wenigen Personen ausreicht, einen Erfolg zu verhindern. Man bedenke dabei, daß hier die Rede von Gruppen mit über 100 Personen ist.

Wenn der verfahrensorientierten Arbeitsteilung im Software Engineering in Zukunft nicht mehr Aufmerksamkeit als bisher gewidmet wird, dann werden die drängendsten Probleme des Software Engineering weiterhin ungelöst bleiben.

7. Ausbildungsdefizite

Die bereits erwähnte Tatsache, daß die Softwareentwicklung auf der Komponentenebene heute ingenieurmäßig beherrscht wird, ist zweifellos eine Konsequenz der hohen Qualität der Ausbildung in diesem Bereich. Wie aber steht es auf dem Gebiet der Engineering–Methoden zur Beherrschung von Systemen der hier betrachteten Komplexität?

Es ist hilfreich, auch hier noch einmal die entsprechende Situation in den traditionellen Ingenieursdisziplinen zu analysieren. In der Engineering Praxis dieser Disziplinen findet man bewährte Organisationsformen und Verfahrensweisen, obwohl diese nicht im Ingenieurstudium vermittelt werden. Am Beispiel veranschaulicht heißt dies, daß die Studenten der Elektrotechnik sehr viel über die Berechnung von Filtern, Verstärkern und Antennen, aber so gut wie nichts über die Projektierung eines Fernsehsendezentrums lernen. Erst später erfahren sie dann in der Praxis, wie ein solches Zentrum projektiert wird. Es ist also offenbar keineswegs nachteilig, den Studienstoff vorwiegend auf die mathematik–orientierten Inhalte zu konzentrieren, weil die anderen Inhalte mühelos in der Praxis nachgelernt werden können und sich ohnehin zum großen Teil einer Vermittlung im Unterricht entziehen.

Auf den ersten Blick scheint es also, als ob kein wesentlicher konzeptioneller Unterschied zwischen der Ingenieursausbildung und der Informatikausbildung bestehe. Diesen Unterschied erkennt man erst beim zweiten Blick. Dann fällt einem nämlich auf, daß man die Ingenieurstudenten in den ersten vier Semestern ihres Studiums überhaupt nichts konstruieren läßt: Dadurch, daß man ihnen zuerst einmal beibringt, wie man Systeme analysieren, modellieren und aspektgebunden darstellen kann, erzeugt man bei ihnen das Bewußtsein für die Probleme und Möglichkeiten der Komplexitätsbeherrschung. Außerdem wird man dadurch der Tatsache gerecht, daß die Ingenieure später mehr Zeit mit dem Analysieren verbringen werden als mit dem Konstruieren. Denn konstruieren wird der Einzelne immer nur einen kleinen Teil eines komplexen Systems, aber er muß dazu die Komponenten, aus denen er seinen Teil aufbauen soll, und die Umgebung, mit der sein Teil kommunizieren soll, vorher verstanden haben. Dies wird ihm meistens erst nach einer eigenen Analyse möglich sein.

Im Gegensatz hierzu wird in der Informatikausbildung das Konstruieren überbetont, während das Analysieren und Modellieren zu kurz kommt. Bereits in der Schule hat der spätere Informatikstudent seine ersten Programme gebastelt; in den ersten Wochen seines Studiums läßt man ihn dann einfache Programme "konstruieren", und im Laufe seines Studiums werden seine Programme immer komplexer, so daß er dann in dem Glauben abgehen kann, er habe die höchste Stufe der Komplexitätsbeherrschung erreicht. Wollte man die Ingenieurausbildung in entsprechender Weise gestalten, dann müßte man dem Elektrotechnik–Studenten, der als Schüler bereits eine Steuerung für seine elektrische Eisenbahn gebastelt hat, in den ersten Wochen seines Studiums einen kleinen Verstärker konstruieren lassen, und er müßte dann im Laufe seines Studiums immer komplexere Apparate konstruieren.

Die derzeitige – weltweit übliche – Informatikausbildung ist keineswegs geeignet, die kritischen Defizite im Software Engineering bewußt zu machen, sie verhindert eher das Entstehen

des erforderlichen Problembewußtseins. Dem Informatikstudenten wird nämlich suggeriert, der Übergang vom "programming in the small" zum "programming in the large" könne im wesentlichen durch die Hinzunahme einiger angemessener programmiersprachlicher Konstrukte bewältigt werden. Niemand weist ihn darauf hin, das die schwierigsten Probleme des "large" gar keine "programming"–Probleme sind.

Solange man in den ersten Semestern der Informatikausbildung nicht auf das Programmieren verzichtet zugunsten des Modellierens und Darstellens informationeller Strukturen, kann aus den Studenten nichts anderes werden als fortgeschrittene Programmierer. Und von diesen darf man nicht erwarten, daß sie die Lösung der speziellen Software Engineering Probleme, welche aus der Systemkomplexität erwachsen, woanders suchen als in noch fortschrittlicheren Programmen. Dort sind sie aber nicht zu finden.

Literatur

[BRO 75] Brooks, F.P. Jr.: The Mythical Man–Month. Addison Wesley Publishing Co., New York, 1975

[BRO 86] Brooks, F.P. Jr.: No Silver Bullet – Essence and Accidents of Software Engineering. IFIP 86, H.–J. Kugler (ed.), S. 1069 – 1076 (1986)

[DEN 91] Denert, E.: Software–Engineering. Springer, Berlin, 1991

[HEN 89] Proceedings of 3rd ACM Software Engineering Symposium on Practical Software Development Environments. P. Henderson (ed.), New York, N.Y., 1989

[HOA 86] Hoare, C.A.R.: The Mathematics of Programming. Clarendon Press, Oxford, 1986

[LUD 91] Ludewig, J.: Software Engineering und CASE – Begriffserklärung und Standortbestimmung. Informationstechnik it 33, H.3, S. 112 – 120, (1991)

[MOR 90] Proceedings of ACM International Workshop on Formal Methods in Software Development. M. Moriconi (ed.), New York, N.Y., 1990

[NAG 90] Nagl, M.: Softwaretechnik: Methodisches Programmieren im Großen. Springer, Berlin, 1990

[WEB 92] Weber, H.: Die Software–Krise und ihre Macher. Springer, Berlin, 1992