

## Wege aus der Softwarekrise

Peter Liebers These: Das Dilemma bei der Softwareentwicklung liegt in der klassischen Pareto-Regel begründet: Nach 80 Prozent der Zeit sind nur 20 Prozent der Software fertiggestellt.



„Anders formuliert: Wenn der Abgabetermin da ist, sind nur 80 % der Software fertig, aber die letzten 20 % brauchen noch 80 % der Zeit.“  
 (Autor Peter Lieber)

RUND 40 PROZENT DER UNTERNEHMEN setzen heute auf Software, die intern selbst entwickelt und auf die spezifischen Anforderungen abgestimmt wird. Zu diesem Ergebnis kam kürzlich eine vom Zentrum für Europäische Wirtschaftsforschung präsentierte Umfrage. Demnach nutzen zwar rund zwei Drittel der Unternehmen aus dem IT- und Mediensektor und dem verarbeitenden Gewerbe herkömmliche Softwarelösungen, die standardisiert angeboten werden. Trotzdem nehmen die Eigeninitiative zu, meinen die Wissenschaftler. „Einer der Hauptgründe für diese prozentuale Verteilungsverschiebung ist, dass die auf dem Markt befindlichen Standard-Softwarepakete die individuellen Anforderungen der Unternehmen häufig nicht mehr passgenau erfüllen“, erklärt Bettina Müller vom ZEW.

Die Zahlen belegen den Trend, dass mit zunehmender Größe der Anteil der Unternehmen, die Software einsetzen, zunimmt. Schließlich steigt der Steuerungs- und damit verbunden der Abstimmungsbedarf zwischen den einzelnen Firmensteilen mit der Größe massiv an. Vor allem Systeme zur Verwaltung von Kundendaten (CRM-Applikationen) und des Mitteleinsatzes (ERP-Systeme) sowie unternehmensübergreifende Software wie SCM-Systeme, die den Datenaustausch zwischen allen an einer Wertschöpfungskette beteiligten Unternehmen koordinieren, lassen sich vielerorts finden.

Angesichts der branchenspezifischen Spezialisierungen der Unternehmen steigen

somit auch die Anforderungen an die zu verwendende Software. Und spätestens an diesem Punkt kommt es zu einem Phänomen, das nun bereits beinahe ein halbes Jahrhundert erstaunlich „jung“ geblieben ist: zur „Softwarekrise“, die erstmals in den 1960er Jahren aufgetreten ist, als die Kosten für die Software jene für die Hardware überstiegen und es in der Folge zu den ersten großen gescheiterten Softwareprojekten kam. Man erkannte, dass die bisher genutzten Techniken mit dem Umfang und der Komplexität von Software nicht Schritt gehalten hatten.

Auf einer NATO-Tagung 1968 in Garmisch-Partenkirchen wurde das Problem diskutiert und als Reaktion der Begriff des Software Engineering geprägt. (Eine der ersten gesicherten Erwähnungen der Softwarekrise findet sich in Edsger W. Dijkstras Dankesrede zum Turing-Preis The Humble Programmer (deutsch: „Der bescheidene Programmierer“, EWD340) die er 1972 hielt und die im Magazin „Communications of the ACM“ veröffentlicht wurde.)

Wie schon aus Dijkstras über 35 Jahre alten Ausführungen zu erkennen ist, kann die Softwarekrise auch heute nicht als beendet betrachtet werden: Die Komplexität der Softwareysteme steigt weiter und damit die Probleme, auch wenn es in der Modernisierung und Strukturierung des Softwareentwicklungsprozesses große Fortschritte gab. Selbst einfache Programme mit einer Länge von unter 500 Programmcodezeilen können ja derart komplex aufgebaut sein, dass sie mathematisch sehr schwer beschreibbar und aufgrund der hohen Zahl von Permutationen (also der Vielzahl von Softwarezuständen) schwer testbar sind. Die Kosten für die Entwicklung und das Testen von Software können mit der Entwicklungszeit exponentiell steigen. Dadurch wird es schwierig, Termine bei der Softwareentwicklung einzuhalten; der Zeitdruck erhöht sich, Programmfehler treten häufiger auf.

Unzufriedene Anwender und schlechte Wartung durch Ressourcenknappheit und

die Unmöglichkeit, Anforderungen zu erfüllen, können die Folgen sein.

Neben den prinzipiellen Ursachen der Softwarekrise, die eher zur Theoretischen Informatik zählen, tragen auch Probleme der Qualitätssicherung zum Scheitern von Softwareprojekten bei. So entsteht Zeitdruck, wenn von fachfremdem Personal Termine vorgegeben werden. Mangelnde Qualitätssicherung, schlechte oder übertriebene Projektorganisation sind genauso verantwortlich für das Misstraten von Projekten wie die ungenügende Einbeziehung des Anwenders oder Kunden. Auch die unzureichende oder überdimensionierte Standardisierung kann wesentlich dazu beitragen.

Die Konzepte der objekt- und der aspektorientierten Programmierung sowie verschiedene Entwicklungsprozesse sollen die Auswirkungen der gestiegenen Komplexität mildern. Ein verstärkter Rückgriff auf erprobte Komponenten und Softwarebibliotheken nutzt bereits geleistete Entwicklungsarbeit effizienter. Der Einsatz von Code-Generatoren und modellgetriebener Soft-

### Schneller abarbeiten!

Die iterative Programmierung (engl. to iterate = wiederholen) verwendet im Gegensatz zur rekursiven Programmierung keine Selbstaufrufe, sondern Schleifen (Wiederholungen von Anweisungen oder Anweisungsfolgen). Dadurch werden die typischen Stapelüberläufe verhindert, und die Programme werden schneller abgearbeitet, da die Methodenaufrufe (mit Kontextsicherung) entfallen.

Die rekursive Programmierung ist aber weniger aufwendig. Prinzipiell lassen sich rekursive Algorithmen auch iterativ implementieren; die rekursive Programmierung ist jedoch vielfach einfacher und leichter verständlich. Ein Beispiel für die iterative Programmierung ist ein Datenbankdurchlauf.

wareentwicklung (ersetzt die Fehlerklasse der zufälligen durch leichter zu findende systematische Fehler) sowie das von Donald Knuth vorgeschlagene Literate Programming sind weitere Möglichkeiten. Eine nicht zu vernachlässigende Quelle von unzureichenden Ergebnissen ist mangelnde Kommunikation zwischen Programmierern und Nutzern.

In diesem Zusammenhang habe ich zehn „Lieber-Regeln“ formuliert, die das Problem – zugegebenermaßen ein wenig polemisch – sprachlich eingrenzen:

#### Die zehn Lieber-Regeln

**1)** Wenn man in ein Softwareprojekt spät zusätzliche Leute einbindet, wird es NOCH später fertig.

**2)** Mehr Geld motiviert zwar das liefernde Unternehmen durchzuhalten, aber nicht die Softwareentwickler.

**3)** Pönalen fördern nur Angst und Aufschläge, aber nicht den Projekterfolg.

**4)** Die Wahrscheinlichkeit, in einem Softwareprojekt einen Prozess zu gewinnen oder zu verlieren, ist geringer als ein Lotto-Sechser! In der Regel werden wegen fachlicher Inkompetenz der Gerichte so lange Sachverständige bestellt, bis die Pro-

zesskosten den Streitwert bei Weitem übersteigen und alle gern einem Vergleich zustimmen. Zusätzlich wird dabei noch auf Zeit gespielt und Zeit = Geld.

**5)** Software spielt für viele Unternehmen eine existenzielle Rolle, wird aber nicht in dem Ausmaß wertgeschätzt, wie es die Software und deren Lieferant verdienen würde.

**6)** Fehler in der Softwareentwicklung (schlechte Dokumentation, schlechte Usability, schlechte Feasibility etc.) führen zu hohen Supportkosten und vielen Gewährleistungsfällen (die teilweise keine sind, aber auch nicht gezahlt werden).

**7)** Die Aufwandschätzung für die Umsetzung eines Softwareteils hängt stärker vom Kunden ab als vom Entwickler – z. B. gilt bei kompetenten Großkunden (die selbst Softwareentwicklung betreiben oder eine IT-Abteilung haben), dass die Entwicklerschätzung (eines „Einzelkämpfers“) im Mittel mit dem Faktor 2,3 multipliziert werden muss, um den tatsächlichen Aufwand zu kennen. (Das hängt natürlich von vielen Faktoren ab, soll aber hier exemplarisch formuliert werden.)

**8)** Wenn ein Entwickler sagt, dass etwas fertig getestet ist, hat es der Entwickler getestet und nicht der Anwender, der es dann auch benutzen muss.

**9)** Wenn ein Entwickler einen Termin nennt, ist es jener Termin, von dem er glaubt, dass er fertig ist, und nicht der Termin, an dem etwas fertig ist.

**10)** Nennt der Entwickler keinen Termin, wird das Softwareprojekt nie fertig. Geht man vom Spannungsfeld „Budget, Termin und Ressourcen“ aus, in dem sich das Softwareprojekt bewegt, werden bei Endkunden oft Pauschalpreise angegeben. Damit kommt der Anbieter dem Kunden entgegen, indem er auf eine für Laien unverständliche Berechnungsbasis verzichtet. Denn der Kunde hat meist keine Vorstellung, was er mit einer Anfrage auslöst. Einerseits ist es die Arbeit, die das System verrichtet, andererseits müssen die gewünschten Informationen auch verfügbar sein. Dazu braucht man Gebäude, Energie, Rechner, Speicher, Software und Menschen, die den Betrieb aufrecht erhalten.

Die Berechnung von IT-Dienstleistungen nach CPU-Minuten ist eine ungenaue Methode, da sie die tatsächliche Rechenleistung nicht berücksichtigt.

Denn „CPU“ (Central Processing Unit) ist kein fester Begriff. Es gibt schnelle und langsame, starke und schwache CPUs, vergleichbar mit den PS beim Automotor – während einer nur 50 PS an die Achse bringt, schafft der andere 100 PS.

#### Der Lösungsansatz mit Zufriedenheitschancen

Am fairsten ist wohl die Abrechnung nach Projektaufwand. Der Kunde bekommt, was er wirklich will, und kann die Ergebnisse prüfen, so oft es passt, weil er ja auch die Zeit dafür bezahlt. Das dafür am besten geeignete Vorgehensmodell sind iterative Methoden wie z. B. Agile Softwareentwicklung, RAD, FDD, Scrum etc. Hier geht es darum, den Softwareentwicklungsprozess flexibler und schlanker zu machen als das bei den „klassischen“ Vorgehensmodellen der Fall ist. Man möchte sich mehr auf die zu erreichenden Ziele fokussieren und auf technische und soziale Probleme bei der Softwareentwicklung eingehen. Die Agile Softwareentwicklung ist eine Gegenbewegung zu den schwergewichtigen und bürokratischen traditionellen Softwareentwicklungsprozessen wie Rational Unified Process oder dem V-Modell.

#### Agile Werte

Das „Agile Manifest“ aus dem Februar 2001 zeigt Wege aus der Softwarekrise:

**1)** Individuen und Interaktionen gelten mehr als Prozesse und Tools.

Zwar sind wohldefinierte Entwicklungsprozesse und hoch entwickelte Entwicklungswerkzeuge wichtig, wesentlich wichtiger sind jedoch die Qualifikation der Mitarbeitenden und effiziente Kommunikation.

**2)** Funktionierende Programme gelten mehr als ausführliche Dokumentation. Gut geschriebene Dokumentationen können zwar hilfreich sein, das eigentliche Ziel der Entwicklung ist jedoch die fertige Software.

**3)** Die stetige Zusammenarbeit mit dem Kunden steht über Verträgen.

**4)** Mut und Offenheit für Änderungen stehen über dem Befolgen eines festgelegten Plans. Im Verlauf eines Entwicklungsprojekts ändern sich viele Anforderungen und Randbedingungen ebenso wie das Verständnis des Problemfeldes. Das Team muss darauf schnell reagieren können.

**In diesem Sinne:  
Raus aus der Softwarekrise!** <<

## Mehr Erfolg mit weniger Aufwand!

### Die Pareto-Verteilung

Die stetige Wahrscheinlichkeitsverteilung wurde nach dem italienischen Ingenieur, Soziologen und Ökonomen Vilfredo Pareto (1848–1923) benannt.

Die Pareto-Verteilung beschreibt das statistische Phänomen, wenn eine kleine Anzahl von hohen Werten einer Wertemenge mehr zu deren Gesamtwert beiträgt, als die hohe Anzahl der kleinen Werte dieser Menge.

Pareto untersuchte die Verteilung des Volkvermögens in Italien und fand heraus, dass ca. 20 % der Familien ca. 80 % des Vermögens besitzen. Banken sollten sich also vornehmlich um diese 20 % der Menschen kümmern, und ein Großteil ihrer Auftragslage wäre gesichert. Daraus leitet sich das Pareto-Prinzip ab, auch „80-zu-20-Regel“, „80-20-Verteilung“ oder „Pareto-Effekt“ genannt. Es besagt, dass sich viele Aufgaben mit einem Mitteleinsatz von ca. 20 % so erledigen lassen, dass 80 % aller Probleme gelöst werden.