Über sieben Brücken musst Du geh'n

Eine Kritik am Software-Engineering

Jens Coldewey
Coldewey Consulting
Curd- Jürgens- Str. 4
81739 München
http://www.coldewey.com
Jens.coldewey@acm.org

Noch bis vor wenigen Jahren galt das Software Engineering als goldener Weg, um die Problem der Software-Entwicklung in den Griff zu bekommen. Bei allen guten Ideen und Ansätzen dieser Richtung mehren sich aber doch die Stimmen, die das ingenieursmäßige Vorgehen in Frage stellen. Für uns Berater stellt sich hier schnell die Gretchenfrage. In dem Artikel werden die wichtigsten Kritikpunkte vorgestellt und Alternativen angerissen.

Früher war die Programmierung von Computern eine Kunst. Virtuos jonglierten die Damen und Herren in ihren weißen Mänteln mit Registern, Lochkarten und dem immer viel zu kleinen Speicher, um beachtliche Leistungen zu bringen. Höhepunkt dieser Techniken war wohl die Mondlandung, die mit Ressourcen gemeistert wurde, mit denen heute kaum mehr ein Maustreiber zurecht käme. Damals wurden Programme üblicherweise in Assembler geschrieben, die Programmierer mussten die Maschine genau kennen, jede Möglichkeit wurde genutzt, Speicherplatz oder Rechenzeit zu sparen. Die Systeme dienten anfangs meist mathematischen Aufgaben, später kamen einfache Datenbanken und Steuerungssysteme hinzu.

In den 60er Jahre wurden dann höhere Programmiersprachen modern. Statt der Maschine mühsam jeden einzelnen Schritt vorzuschreiben, wurden Sprachen entwickelt, die vom Menschen leichter verstanden wurden. Sprachen wie ALGOL-60 und später ALGOL-68 und FORTRAN deckten erfolgreich mathematische Aufgabenstellungen ab, COBOL setzte sich bald für kaufmännische Anwendungen durch. Die Beherrschung einer bestimmten Programmiersprache wurde zu einer Schlüsselqualifikation in der DV-Branche. Schon bei der Entwicklung von COBOL wurde damals ein Anspruch erhoben, der bis heute nicht wirklich eingelöst wurde: Die Programmierung der Computer sollte den Fachbereichen selbst möglich sein, die Abhängigkeit von den Programmierern sollte endlich vorbei sein. Seitdem dieser Anspruch das erste Mal formuliert wurde, hat sich das Programmieren - oder die Software-Entwicklung wie es heute heißt - zu einem der tragenden Zweige der westlichen Wirtschaften gemausert, einschlägige Firmen wie Microsoft, Oracle oder SAP sind Schwergewichte in den jeweiligen Börsenindizes. Alteingesessene Wirtschaftsnationen zittern vor der Kompetenz von Schwellenländern und weltweit verdienen viele Millionen Frauen und Männer ihren Lebensunterhalt mit Programmieren. Längst haben die Sozialwissenschaftler die nächste industrielle Revolution ausgerufen, die Abhängigkeit von der programmierenden Zunft ist mittlerweile total.

Die Systeme wurden immer leistungsfähiger — und immer komplexer. Bald wuchs die Software ihren Erstellern über die Köpfe, die Entwicklungsmannschaften wurden immer größer und die Fehlerraten immer eindrucksvoller. Bald wurde die "Software Krise" ausgerufen, es mußte etwas passieren! Es wurde mehr und mehr klar, dass die Entwicklung der Software nicht nur virtuoses Programmieren möglichst ausgefeilter Algorithmen umfasst, sondern die Beherrschung der Komplexität eine immer wichtigere Rolle spielt. Zwei der wichtigsten Impulse dazu kamen von Männern, die unterschiedlicher kaum sein können: Edsger Dijkstra und Alan Kay.

Edsger Dijkstra, meines Wissens ursprünglich Numeriker, war nicht nur federführend an der Entwicklung der strukturierten Programmierung beteiligt, sondern entwickelte auch die sogenannten "Abstrakten Datentypen", aus denen sich die Prinzipien der Modularisierung und Datenkapselung

Abgedruckt in OBJEKTSpektrum 1/2001

ableiten. Die Idee ist einfach: Man zerlege das komplizierte System in kleine, übersichtliche Teile, die kein Wissen über gegenseitige Interna besitzen und über genau definierte Schnittstellen miteinander kommunizieren. Damit war das wichtigste technische Hilfsmittel geschaffen, um die immer größer werdenden Systeme der 70er und 80er Jahre nach im Griff zu behalten.

Alan Kay, damals noch ein junger Forscher an den Xerox Laboratorien, ging einen anderen Weg. Er verglich Software mit den komplexesten Systemen, die uns bekannt sind, mit Lebewesen. Alle höheren Lebewesen bilden sich aus Zellen. Jede Zelle hat einen bestimmten Zelltyp, und ist auf eine bestimmte Aufgabe spezialisiert. Von jedem Zelltypen gibt es in einem Körper abertausende Exemplare, die zwar weitgehend gleich aufgebaut sind, aber doch ihre eigene Identität besitzen. Die Zelle ist von der Außenwelt durch eine Zellmembran geschützt und kommuniziert mit dieser Umwelt über genau definierte Kanäle, aber ohne diese genauer zu kennen. Auch wenn Zellen über ihre gesamte Lebenszeit ihren Typ beibehalten, so sind sie doch in der Lage, neue Zellen zu erzeugen, die leicht angepaßte Eigenschaften besitzen. In Ermangelung einer besseren Metapher nannte Kay ein entsprechendes Pendant in der Software "Objekt" und schuf damit die Basis für die objektorientierte Programmierung.

Diese beiden Ideen zeigen bei flüchtiger Betrachtung einige deutliche Parallelen. Insbesondere die Idee des geheimen Inneren, das nur über definierte Schnittstellen mit dem Außen kommuniziert, scheint beiden Ansätzen gemeinsam. Nicht übersehen darf man dabei aber, dass beide Ansätze völlig unterschiedlichen Philosophien folgen: Dijkstras Ansatz ist in bester westlicher Wissenschaftstradition analytisch: Ein gegebenes, großes System wird solange in kleinere Teilsysteme zerlegt, bis die einzelnen Bestandteile beherrschbar sind — Cäsars Prinzip "teile und herrsche" findet sich hier wieder. Kay hingegen schwebt eher die Synthese vor: Aus vielen kleinen Einheiten setzt sich ein großes System zusammen. Idealerweise existieren diese Einheiten bereits zuvor und können sich selbst an ihre neue Umgebung anpassen. Es ist müßig zu erwähnen, dass der Großteil der Softwareprojekte heute eher dem analytischen Ansatz folgt, während Kays Synthese noch immer visionär erscheint, auch wenn so manche Veröffentlichung über Komponenten gerne den gegenteiligen Eindruck erweckt. Mit verantwortlich dafür ist wohl auch der Umstand, dass bis heute kaum eine gängige Programmiersprache die Ideen Kays auch nur annäherungsweise umsetzt. Am ehesten kann meines Erachtens Smalltalk noch ein Gefühl dafür vermitteln, wohin ein solcher Ansatz führen könnte.

Die Wurzeln des Software-Engineering

Neben den Verbesserungen der Programmiertechniken hielt nämlich noch ein ganz anderer Ansatz Einzug in die Software-Entwicklung: die Idee, Software ingenieursmäßig zu entwikkeln. Warum sollte man Software nicht entwickeln können, wie man etwa ein Chemiewerk baut? Hier werden zunächst die Anforderungen gesammelt, dann werden Pläne zunehmender Granularität erstellt, bis schließlich die Bauaufträge an die einzelnen Spezialfirmen erteilt werden, die dann die funktionierende Fabrik bauen. Konsequent umgesetzt sollte es möglich sein, Software mit der gleichen Zuverlässigkeit und Kostentreue herstellen zu können, wie heute Häuser und Fabriken gebaut werden. Wie geplagte Bauherren möglicherweise wissen, verfällt nach deutschem Standesrecht zum Beispiel die Honorarforderung eines Architekten, wenn die Baukosten den Voranschlag um mehr als 30% überschreiten. Eine solche Regelung würde- übertragen auf die Softwareindustrie- sicherlich zum Ruin der halben Branche führen.

Die Verheißung pünktlicher und kostentreuer Software-Lieferung mit Hilfe von Techniken, die man sich aus der Bauindustrie abschaut, hat einen deutlichen Charme. Dem bis vor wenigen Jahren noch üblichen künstlerischen Chaos bei der Entwicklung von Software wird ein definierter Entwicklungsprozess entgegengestellt, der sicherstellt, dass das System auch pünktlich ausgeliefert wird - zumindest in der Theorie. Prozesse, die dieser Philosophie folgen, definieren eine Reihe von Dokumenten - auch als *Artefakte* bezeichnet - die vor der eigentlichen Programmierung zu erstellen sind. Da gibt es Anforderungsdokumente, Geschäftsprozessanalysen, Grobspezifikation, Feinspezifikation, Oberflächenspezifikation, Testspezifikation, Architektur, Grobdesign, Feindesign, Moduldesign, Schnittstellenentwürfe, Testdesign, Oberflächendesign. In großen Projekten sind diese Dokumente nicht nur für das eigentliche System zu verfassen, sondern auch für die Entwicklungsumgebung, für die Testumgebung, für das Konfigurationsmanagement, für die Einführung und für die Wartung. Was hier fast wie Kabarett klingt, ist lediglich eine komprimierte Sicht auf die heute gängigen Prozesse, wie z.B. das V-Modell (vgl. [BrD95]) oder den Rational Unified Process (vgl. [JBR99]). Wegen der vielen zu erstellenden Unterlagen werden solche Prozesse als "schwergewichtige Prozesse" bezeichnet.

Theorie und Praxis

Meine persönliche Erfahrung mit Kundenprojekten, aber auch als Projektleiter sieht da leider etwas anders aus. Projekte, die mit schwergewichtigen Prozessen arbeiten, zeigen immer wieder einige typische Symptome:

- *Die Dokumente sind schlecht geschrieben*. Wichtige Details werden in ihnen übersehen, Festlegungen sind unpräzise. Klassisch ist zum Beispiel die Anforderung, ein System müsse dialogfähig sein. Daraus werden dann von Auftragnehmer und -geber meist unterschiedliche Schlußfolgerungen gezogen, wie schnell denn das System nun auf bestimmte Anfragen zu antworten hat- eine Quellen langen und unergiebigen Streits. Aber auch Architekturbeschreibungen, die eher Verkaufsprospekten zu 3-*Tier* Architekturen ähneln, als dem hinterher umgesetzten System, sind keine Seltenheit.

- *Die Projekte sind in Verzug*. Gerade wenn engagierte Projektleiter versuchen, möglichst gute Anforderungs- und Analysedokumente zu erstellen, geraten sie bereits in den frühen Projektphasen in Verzug. Werden die Anforderungen anfangs nicht genau genug festgehalen, geraten sie spätestens während Analyse und Design in Zeitprobleme. Häufig wird versucht, den Verzug durch massive Überstunden wieder einzuholen. Es scheint mittlerweile als normal angesehen zu werden, dass zumindest Projektleiter eine 80-Stunden-Woche fahren. Dabei sollte man nicht vergessen, dass jemand, der 80 Stunden pro Woche arbeitet, keinen sozialen oder psychischen Ausgleich mehr zu seinem Beruf hat. Ihm oder ihr fehlen also alle Puffer zur Bewältigung von Stress und schwierigen Situationen. Unter solchen Bedingungen auch noch seine soziale Kompetenz zu erhalten, grenzt an eine Aufgabe von herkulischem Ausmaß.

- *Im gesamten Team werden massive Überstunden geleistet*. Bisher habe allen meinen Kunden, die unter diesem Symptom litten, Ed Yourdons Meisterwerk “Death March” empfohlen, das die wohl beste Analyse dieses Symptoms enthält, die ich bisher gelesen habe (siehe [You97]).

- *Gerade in objektorientierten Projekten ist die Design- und Code-Qualität häufig katastrophal.* Von Wiederverwendbarkeit kann keine Rede sein, die Systeme haben erhebliche Stabilitäts- und Performanceprobleme.

- *Die Auslieferung erfolgt meist zu spät und dann auch nicht in vollem Umfang*. Oft schließen sich lange Streitereien zwischen Auftragnehmer und Auftraggeber über die Einschätzung von Fehlern und kostenlose Nachbesserungen an. Für keinen der Beteiligten ist das eine besonders erquickliche Situation.

Ich vermute, dass es kaum einen Leser gibt, dem diese Symptome nicht irgendwie bekannt vorkommen. Aber wenn uns der Einsatz von Software-Engineering hilft, die Projekte planbar und kontrolliert durchführbar zu machen, warum laufen dann so viele Projekte so schrecklich daneben? Als ich mein erstes Projekt selber leitete, war die Sache für mich klar: Die anderen wendeten die Techniken halt nicht konsequent genug an. Also bestand ich darauf, dass saubere Anforderungsdokumente erstellt werden, es wurde ein hundertseitiges Architekturpapier geschrieben, Testfallspezifiktionen, Werkzeuge, um aus allen diesen Dokumenten automatisch Code zu erzeugen und ein ausgefeiltes Konfigurationsmanagement aufgebaut. Nach zwei Jahren bedankte sich der Auftraggeber für meine Bemühungen und entzog meinem Arbeitgeber den Auftrag. In dieser Zeit waren tolle Konzepte entstanden, aber noch keine Zeile Code für das Endsystem, weil der verwendete Prozess die Codierung erst für eine spätere Phase vorgesehen hatte. Natürlich fehlte es auch nicht an hilfreichen Kollegen, die mir mit guten Ratschlägen zur Seite standen:

“Bei hundert Seiten Architektur hat es halt am nötigen Pragmatismus gefehlt.” Aber worin hätte der Pragmatismus bestanden? Ohne Architektur starten? Wohl kaum! “Die Objektorientierung ist halt nicht ausgereift für große Projekte.” Nicht besonders hilfreich, wenn man bedenkt, dass die vorgesehene grafische Oberfläche mit funktionaler Technik überhaupt nicht machbar gewesen wäre.

Aus heutiger Sicht lag mein Fehler wirklich an zu großem Perfektionismus, aber nicht in der Ausarbeitung des Designs, sondern beim Verfolgen eines schwergewichtigen Wasserfall-Prozesses.

Der Angriff auf die Grundfesten

Das Projekt ist gescheitert, *weil* ich mich an den Prozess gehalten habe? Diese Aussage ist provokant und bedarf daher einer genaueren Erklärung. Eine der Grundannahmen des klassischen Software-Engineering betrifft die Fehlerkosten. Je früher im Prozess ein Fehler gemacht wird, um so teurer wird er im Laufe des Projekts. Oder anders formuliert: Was du am Anfang sagst, muss richtig sein, weil Du es nicht mehr ändern kannst. Und weil diese Aussage so wichtig ist, noch eine dritte Formulierung des gleichen Axioms: Einmal geschriebene Software ist nicht mehr änderbar. Die meisten Prozesse sind daher darauf

ausgelegt, Fehler möglichst früh zu vermeiden und alle Fehler auszumerzen, bevor die Programmierung beginnt. Bei dem Entwicklungsprozess eines Kunden habe ich einmal 211 Reviews unterschiedlichster Dokumente gezählt, bevor die Programmierung starten konnte. Meines Wissens ist dieser Ansatz nie umgesetzt worden.

Genau betrachtet leitet sich aus dem Axiom der„Nicht-mehr-Änderbarkeit“ die Parallelität zwischen der Bauindustrie und dem Software-Engineering ab: Ein chemischer Reaktionsturm, der einen Meter zu weit links gebaut wird, kann die gesamte Fabrik hinfällig machen. Ihn nachträglich zu verschieben kann Millionen kosten. Aber gilt das wirklich auch für Software? Was wäre, wenn Software einfach änderbar wäre? Wie sähe ein Prozess aus, wenn man keine Angst mehr vor Änderungen haben müsste?

Ballast abwerfen

Zunächst einmal müsste man versuchen, alle Änderungshindernisse zu beseitigen, die aus dem Prozess selbst kommen. Sind 211 Dokumenttypen erst einmal geschrieben, verbietet sich jede Änderung von selbst, weil dann ja alle 211 Dokumenttypen konsistent nachgezogen werden müssten. Will ich leichte Änderbarkeit erreichen, muss ich also möglichst wenig Quellen haben, im besten Fall sogar nur eine einzige. Da für ein lauffähiges System zumindest Code notwendig ist, sollte man also versuchen, weitere Dokumente möglichst zu vermeiden, ohne dabei in die gute alte Zeit des chaotischen Hackens zurück zu fallen. Dies ist die Grundidee der sogenannten “leichtgewichtigen Prozesse”. Die beiden bekanntesten Vertreter sind der „Crystal Clear Prozess“ von Alistair Cockburn (vgl. [Coc02]) und das “Extreme Programming” von Ward Cunningham, Kent Beck und Ron Jeffries (vgl. [Bec00]).

Zur Erklärung möchte ich hier aus Alistair Cockburns “Manifest der Softwareentwicklung” zitieren[1]:

“Softwareentwicklung ist ein kooperatives Spiel, in dem die Mitspieler Markierungen und Requisiten verwenden, um sich gegenseitig und selbst zu erinnern und über den nächsten Spielzug zu informieren und zu inspirieren. Das Ziel des Spiels ist das lauffähige Softwaresystem; die Überbleibsel des Spiels sind eine Reihe von Markierungen, welche die Spieler des nächsten Spiels informieren und ihnen helfen. Das nächste Spiel ist die Änderung oder der Ersatz des Systems, aber auch die Schaffung eines Nachbarsystems.”

Dieses Manifest im Detail zu analysieren würde den Rahmen dieses Artikels sprengen, daher sei hier lediglich auf die in der Fußnote angegebene Web-Adresse verweisen. Ich möchte hier nur im Folgenden Aspekte herausgreifen, die meines Erachtens für das Verständnis leichtgewichtiger Prozesse von Bedeutung sind.

[1] Die englische Originalfassung ist verfügbar unter http://members.aol.com/acockburn/manifesto.html. Dort findet sich auch eine ausführliche Diskussion des Manifests. Die Übersetzung und Verwendung hier erfolgte mit freundlicher Genehmigung von Alistair Cockburn.

Softwareentwicklung ist ein kooperatives Spiel

Im Gegensatz zu den üblichen Wettkampfspielen gibt es bei kooperativen Spielen keine Gewinner oder Verlierer, sondern die ganze Gruppe arbeitet auf ein gemeinsames Ziel hin. Klettern oder Musizieren sind solche Spiele, aber auch der große Fundus an “New Games”, die in allen Spielarten der Pädagogik zu finden sind. Unabhängig von dem Ziel, das es zu erreichen gilt, ist diesen Spielen eines gemeinsam: Sie erfordern ein hohes Maß an Kommunikation und Interaktion zwischen den Spielern und der Erfolg hängt von jedem einzelnen ab. Bei manchen dieser Spiele, wie zum Beispiel beim Klettern, hängt sogar das Leben der Mitspieler am reibungslosen Zusammenspiel aller Beteiligten.

Auch in der Softwareentwicklung hängt der Erfolg des Projektes von der reibungslosen Kommunikation zwischen den Beteiligten ab. Wenn die Entwickler die Probleme der Anwender nicht verstanden haben, ist das Projekt zum Scheitern verurteilt. Das gleiche gilt, wenn sie die Architektur nicht begriffen haben. Um diese Kommunikation sicherzustellen, setzen schwergewichtige Prozesse auf Dokumente, die geschrieben, einem Review unterzogen und schließlich ausgetauscht werden. Allerdings sind geschriebene (oder auch gezeichnete) Dokumente eine sehr schmalbandige Art, zu kommunizier.en. Um nützlich zu sein, müssen Sie sehr präzise und unmissverständlich sein. Leider ist der dafür erforderliche Aufwand in vielen Projekten unwirtschaftlich und wird deswegen nicht betrieben.

Leichtgewichtige Prozesse setzen dagegen auf direkte zwischenmenschliche Kommunikation im Gespräch oder in Teammeetings. Das bringt mindestens drei Vorteile:

- Es ermöglicht sofortige Rückfragen.
- Alle Beteiligten können zu einem sehr frühen Zeitpunkt einbezogen werden.
- Die gesamte Bandbreite der Kommunikation kann genutzt werden, von der Körpersprache über Tafelskizzen bis hin zur gemeinsamen Arbeit an Entwicklungswerkzeugen.

Erkauft werden diese Vorteile durch schlechtere Kontrollierbarkeit und eine Größenbegrenzung des Teams, auf die ich später noch komme werde.

Ziel ist das lauffähige Softwaresystem

Was wie eine Selbstverständlichkeit klingt, kann kaum oft genug hervorgehoben werden. Vielleicht sollte man zur Verdeutlichung schreiben: Ziel ist *einzig* das lauffähige Softwaresystem. Ich habe dutzende von Projekten gesehen, in denen das Ziel augenscheinlich darin bestand, eine tolle Entwicklungsumgebung zu schaffen, schöne Dokumente zu schreiben, oder eine besonders leistungsfähige Middleware zu bauen. Für Firmen, die Entwicklungsumgebungen oder Middleware als Produkte bauen, oder für Autoren sind das die richtigen Ziele. Bei den meisten Projekten verdient aber der Auftraggeber nicht sein Geld mit dem Vertrieb von Entwicklungsumgebungen, Middleware oder Büchern. Die Software ist ein Werkzeug für ihn, das vor allem Geld kostet. Sie muss so schnell, wie möglich in Produktion gehen, weil dadurch ein Wettbewerbsvorteil entstehen könnte (oder ein Wettbewerbsnachteil ausgemerzt wird), und sie sollte auch möglichst wenig Folgekosten in Betrieb und Wartung produzieren. Warum Millionenbeträge in die Eigenentwicklung technische Infrastruktur oder (meist unlesbarer) Dokumente investieren, wenn ich mittlerweile sehr gute Infrastrukturen kaufen kann und die Dokumente durch intensive Kommunikation im Team weitgehend reduzieren kann? Wenn es die benötigte Infrastruktur auf dem Markt noch nicht gibt, kann man häufig durch geringfügiges Abspecken der Anforderungen auf käufliche Umgebungen ausweichen.

In dem von mir oben erwähnten Projekt war zum Beispiel eine der Anforderungen, die grafische Oberfläche plattformübergreifend zu Bauen. Der Kunde wollte sich die Option offen halten, später noch von Windows auf UNIX umzustellen. Als alter UNIX-Freund nahm ich die Anforderung gerne an und produzierte damit einen Aufwand von mehreren Bearbeiterjahren, da wir viele Entwicklungswerkzeuge nicht mehr nutzen konnten und zum Teil selbst schreiben mussten (Java war damals noch nicht am Markt). Es wäre vermutlich billiger gewesen, im Falle einer Migration das System einfach noch mal neu zu schreiben. “You Aren't Gonna Need It” (auf dt. „Du brauchst es eh’ nicht”) ist eines der schönsten Bonmots von Kent Beck. Neunzig Prozent der Dinge, die als “vielleicht müssen wir später mal...” in Softwareprojekte hineinkriechen, werden niemals benötigt. Wenn man sich diese Aufwände spart und dafür bei jeder Anforderung, die doch noch kommt, den dreifachen Aufwand hat, kommt unter dem Strich noch immer eine Ersparnis von 70 Prozent zustande! Aus meiner persönlichen Erfahrung erscheinen mir bei einer einigermaßen tragfähigen Architektur diese Zahlen durchaus realistisch. Ziel ist also das lauffähige Softwaresystem- und sonst nichts. Gemacht wird nur, was man dafür benötigt. Alles andere kann auf später warten, selbst wenn es dann etwas teurer wird.

Der Zweck heiligt die Mittel

Außer Code dient alles, was innerhalb eines Projekts entsteht, nur dem Zweck, den Teammitgliedern zu helfen und sie zu unterstützen. Mit anderen Worten: Der Zweck heiligt die Mittel. Wenn das Team der Ansicht ist dass es besser arbeitet, wenn ein Klassenmodell mit kleinen, gelben Klebezetteln und Wollfäden an der Wand entstehen lässt, ist das genauso gut, als wenn dafür ein CASE-Werkzeug eingesetzt wird oder das Klassenmodell nur im Code-Browser existiert. Natürlich vertreten Werkzeughersteller hier eine etwas andere Position, doch darf man sie hier getrost als voreingenommen betrachten. Wichtig ist lediglich, dass das Team seine Arbeit machen kann und regelmäßig die Möglichkeit hat, darüber zu reflektieren, wie es die Arbeit macht. Welche Dinge schriftlich kommuniziert werden und auf welche Weise, hängt dabei stark vom jeweiligen Team ab. Es ist kaum vorstellbar, dass eine irgendwie geartete Methodengruppe oder auch ein Projektleiter oder ein Berater hier mehr leisten können, als dem Team Optionen und Alternativen aufzuzeigen. Die Spielregeln kooperativer Spiele müssen von allen Mitspielern entworfen und getragen werden. Das Management muss also die Entscheidung des Teams nach außen durchfechten. Vom Controlling verordnete Klebezettel als billiger Ersatz für CASE-Werkzeuge wirken letal.

“Moment mal,” werden erfahrene Projektleiter und Anwendungsentwickler jetzt einwenden, “Was ist mit der Wartungsdokumentation? Ich kann doch schlecht die Tapete von der Wand reißen und archivieren, nur weil

ein paar Klebezettel darauf mein System dokumentieren." Das ist ohne Einschränkung korrekt. Aber Hand aufs Herz: Wie viele Projekte haben Sie in der Wartung, in denen die Dokumentation wirklich so aktuell ist, dass sie von der Wartungsmannschaft genutzt und aktualisiert wird? Ich würde 1:10 wetten, dass weniger als 10% der Anwendungen in diesem Zustand sind, und könnte mit den Wetterlösen vermutlich einen äußerst attraktiven Fond begründen. In den allermeisten Fällen werden Dokumente früher Phasen zwar geschrieben, aber spätere Änderungen werden nicht mehr eingepflegt, wenn der Terminplan drückt.

"Im Code steckt die Wahrheit" ist eine alte Weisheit, an deren Gültigkeit alle Fortschritte des Software- Engineering wenig geändert haben. Anstatt gegen diese Wahrheit anzukämpfen, kann man sie auch als gegeben hinnehmen. In diesem Fall würde man vielleicht ein Fünftel des Aufwandes, der bei schwergewichtigen Prozessen in die Dokumentation gesteckt wird, in die Vereinfachung des Codes stecken. Was ich an Dokumentation wirklich brauche, ist ein kurzer Überblick über die Funktionsweise und die wesentlichen Elemente des Systems; alles andere sollte über Code-Browser zur Verfügung stehen. Ein solcher Überblick- nennen Sie ihn Architektur, Vision, oder wie immer Sie wollen, - sollte auf zehn Seiten passen oder in einer halben Stunde erklärt sein.

Ehrliche Werbung

"Prima, endlich können wir uns diese lästigen Analyse- und Dokumentationsarbeiten sparen und direkt anfangen, das zu tun, was uns am meisten Spaß macht: Hacken!" Diese Schlussfolgerung ist nicht selten. Ich erinnere mich an eine Diskussion, die ich im Oktober mit Kent Beck hatte, ob der aktuelle Hype um „eXtreme Programming" (XP) den leichtgewichtigen Prozessen mehr Nutzen oder mehr Schaden bringen würde. Beck räumte ein, dass auch er immer wieder "Erfahrungsberichte" hören würde, in denen stolz verkündet wird: "Wir machen XP, aber das Pair Programming lassen wir weg und das Planungsspiel funktioniert bei uns nicht so richtig und das mit den automatischen Tests ist bei uns früher schon mal gescheitert und der Anwender im Team ist uns zwar versprochen worden, kommt aber erst im nächsten Jahr!" Für den geschulten Blick bleibt hier nur eine Diagnose: Da hat jemand die Idee leichtgewichtiger Prozesse nicht richtig verstanden. Sie sind keineswegs eine Aufforderung, wieder in die dunklen Zeiten des Hackens zurückzufallen und alles über Bord zu werfen, was in den letzten zwanzig Jahren entwickelt wurde. Leichtgewichtige Prozesse verlangen vielmehr ein ausgesprochen diszipliniertes Vorgehen, sie besitzen nur sehr wenig Redundanz und sind daher anfälliger gegen Prozessabweichungen als schwergewichtige Prozesse. Dass man bei konsequenter Anwendung dieser Prozesse auch auf manche lästige Dokumentationsarbeit verzichten kann, ist eher die Ernte, die man einfährt für reibungslos funktionierendes Konfigurationsmanagement, voll automatisierte Tests und anderen Maßnahmen.

Was ich in dieser Diskussion also bisher stillschweigend unterschlagen habe, sind drei Voraussetzungen, die absolut notwendig sind, damit ein solches Vorgehen trägt:

- Die Rückkopplungsschleifen müssen kurz genug sein,
- Im Team muss Kommunikation möglich sein.
- Das Team muss seinen Job beherrschen.

Zumindest die beiden ersten Voraussetzungen sind in vielen klassischen Projekten nicht gegeben.

Möglichkeit zu lernen

Kurze Rückkopplungsschleifen benötigen alle Beteiligten, um den Erfolg des Projekts sicherzustellen. Das Team benötigt Erfahrung, um einschätzen zu können, welche Verfahren funktionieren und welche nicht. Das Management benötigt früh Rückkopplung, um sicher zu sein, dass das Projekt nicht aus dem Ruder läuft. Die Anwender brauchen schließlich frühzeitig lauffähige Systeme, um zu sehen, ob das System wirklich das liefert, was sie benötigen. Für alle drei Zwecke ist laufender Code das Mittel der Wahl. Ein komplett durchlaufener Entwicklungszyklus gibt dem Team überhaupt erst die Möglichkeit, die Brauchbarkeit einzelner Verfahren zu beurteilen. Für das Management ist ein laufendes System ein ziemlich unbestechlicher Indikator dafür, dass etwas läuft. Angeblich sind Softwaresysteme zu 80% ihrer Zeit zu 80% fertig. Die restlichen 20% benötigen die anderen 80% der Zeit. Ein laufendes System ist per Definition fertig, kleinere Nachbessererungen dienen dazu, weitere Stufen darauf setzen zu können, und zählen also schon zur nächsten Stufe. Leichtgewichtige Prozesse verlangen also zwingend ein iteratives Vorgehen.

Es fehlen noch die Anwender. Wenn es einem Experten schon schwer fällt, die Brauchbarkeit eines Systems allein anhand von Dokumenten zu beurteilen, um wie viel schwerer tut sich dann ein Laie, der lediglich die Arbeit gemacht haben möchte? Ein laufendes System gibt den Anwendern die be-

sten Möglichkeiten, Änderungswünsche zu formulieren und sie mit den Entwicklern durchzusprechen. Schließlich ist Software änderbar.

Miteinander Reden

Ein anderes Problem ist die Kommunikationsfähigkeit eines Teams. Damit ein Team miteinander reden kann, darf es nicht zu groß sein und muss beieinander sitzen- beides Voraussetzungen, die häufig missachtet werden. Es gibt eine obere Grenze für die Teamgröße, ab der die direkte Kommunikation nicht mehr funktioniert. Mathematisch gesehen steigt die Anzahl der notwendigen Kommunikationsverbindungen mit dem Quadrat der Anzahl der Teammitglieder. Die Zahl, wo der Rubikon überschritten wird, schwankt zwischen 8 und 15 Personen- je nach Quelle. Ab dieser Grenze ist es nicht mehr möglich, alle Teammitglieder auf dem Laufenden zu halten und ohne Rundschreiben, schriftliche Vorgaben und Ähnlichem zu arbeiten. In Abwandelung eines Wortes von Tom DeMarco könnte man sagen, ein Team von 40 Personen ist nur unwesentlich ineffizienter, als eines von zehn. Andere Redensarten stützen diese Überlegung. “Wenn man ein verspätetes Projekt personell aufstockt, wird es noch später” ist eine davon.

Was aber ist mit den vielen Projekten, an denen 100 und mehr Personen arbeiten? Nun, die erschreckend niedrige Erfolgsquote solcher Projekte bestätigt diese Überlegungen eher, als ihnen zu widersprechen. Leider ist damit aber nicht gesagt, dass eine radikale Verkleinerung des Teams immer die richtige Lösung wäre. Manche Systeme, wie zum Beispiel Telekommunkationsanlagen sind einfach nicht mit kleineren Teams zu machen. Man kann natürlich kleine Teilteams bilden und versuchen, die Schnittstelen zwischen ihnen zu minimieren. Aber diese Schnittstellen müssen sauber dokumentiert und schriftlich fixiert sein, sonst läuft das System aus dem Ruder. Dennoch gibt es viele Fälle, wo ein Team von acht oder zehn versierten Softwareentwicklerinnen und -entwicklern bei weitem rentablere Ergebnisse erbracht hätten, als große Teams. Als Paradebeispiel möchte ich nur das Gehaltssystem von Chrysler nennen, das durch eine Schlankheitskur von 26 auf acht Entwickler innerhalb von zwei Jahren in Produktion gesetzt werden konnte.

Neben der reinen Größe ist aber auch die räumliche Nähe des Teams entscheidend. Bei einem meiner Kunden wurden einmal wegen eines Umzugs vier Kolleginnen und Kollegen vom restlichen Team räumlich getrennt. Die Folge waren erhebliche Turbulenzen im Projektablauf. Es dauerte fast drei Monate, bis sich das Team auf die neue Situation eingestellt hatte und verstärkt auf Email und eine Fehlerdatenbank gesetzt hatte. Die Vorstellung, man könne Softwareentwicklung rund um den Erdball verteilen, um die einzelnen Zeitzonen oder einfach die niedrigen Löhne in manchen Ländern besser nutzen zu können, verkennt diesen Kommunikationsaspekt völlig. Wer einmal geklettert ist, weiß wie schwierig das Sichern wird, wenn die kletternde Person auf einmal außer Sichtweite gerät. Oder können Sie sich ein Konzert vorstellen, bei dem die Streicher in Japan sitzen, die Bläser aber in London und die Paukisten in Lima? Versuche in dieser Richtung hatten eher medialen als musikalischen Wert.

Ein dritter Aspekt ist die persönliche Kommunikationsfähigkeit der einzelnen Teammitglieder. Kommunikation ist Handwerk. Zwar lernen wir alle einen guten Teil des Handwerks schon in den ersten Lebensjahren, so wie die meisten Kinder auch Malen oder Schnitzen lernen. Professionelle Maler oder Schnitzer haben aber meist auch eine weitere Ausbildung genossen. Wenn Softwareentwicklung Kommunikation ist, müssten professionelle Softwareentwickler dann nicht auch professionelle Kommunikatoren sein? Wenn ich mir überlege, wie viel Zeit und damit Geld und Aufwand in vielen Projekten einfach durch Missverständnisse und schlechte Kommunikation verloren gehen, ist es ein Wunder, dass viele Unternehmen noch immer das Erlernen von Programmiersprachen als die Schlüsselqualifikation eines Softwareentwicklers sehen und nicht das Erlernen professioneller Kommunikation. Da wird eher auf schlecht geschriebene Dokumente zurückgegriffen, die zu astronomischen Kosten für den Papierkorb geschrieben werden.

Garbage in, Garbage out

Als letzte Voraussetzung fehlt noch die schlichte Qualifikation des Teams. “Und hier habe ich noch zwei Wochen geplant, um das Team in Objektorientierung zu schulen.” Welcher Berater hätte solche Sätze noch nicht gehört? Was würden Sie mit einem Bauleiter machen, der Ihnen erklärt, die Maurer wären gerade noch auf einem zweiwöchigen Kurs über Hausbau? Ich persönlich würde mir schleunigst eine neue Baufirma suchen! Es ist richtig, dass gute Softwareentwickler mit der entsprechenden Grundqualifikation sehr schnell neue Technologien erlernen können. Es ist aber auch richtig, dass schlechtes Design einer der häufigsten Ursachen für gescheiterte Projekte ist. Die Vorstellung, man könne mit Hilfe eines wie auch immer gearteten Prozesses aus einem schlechten Team gute Arbeit holen, ist - vorsichtig gesagt - naiv.

Über sieben Brücken musst Du geh'n

So lautet ein alter Rocktitel der Gruppe Karat. In der Softwareentwicklung haben wir schon einige Brücken hinter uns: Wir haben die Mathematik als unentbehrliche Hilfe beim Finden von Algorithmen erkannt, wir haben Programmiersprachen entwickelt, wir haben gelernt, die Komplexität zu beherrschen, wir haben von anderen Ingenieursdisziplinen das Management großer Projekte gelernt und wir haben unentbehrliche Werkzeuge entwickelt, die uns dabei helfe. Dies reicht von Compilern über Konfigurationsmanagement und automatische Tests bis hin zu kompletten integrierten Entwicklungsumgebungen und zahlreiche Frameworks. Es wird Zeit, dass wir von dieser langen Reise Ballast abwerfen und die nächste Brücke suchen. Vielleicht besteht sie aus der Erkenntnis, dass Software eben keine Fabrik ist, sondern änderbar und dass sie vor allem in einem kommunikativen Prozess erstellt wird.

Autor

Jens Coldewey ist unabhängiger Berater. Er hilft großen Organisationen bei der Einführung und Umsetzung objektorientierter Konzepte. Seine E-Mail-Adresse lautet: jens_coldewey@acm.org.

Literatur

[Bec00] Kent Beck: *Extreme Programming Explained — Embrace Changes;* Addison-Wesley, Reading, Massachusetts, 2000; ISBN 0-201-61641-6

[BrD95] Adolf-Peter Bröhl, Wolfgang Dröschel: *Das V-Modell — Der Standard für die Softwareentwicklung mit Praxisleitfaden*; R. Oldenbourg Verlag München, Wien, 1995; ISBN 3-486-23470-6

[Coc02] Alistair Cockburn: *Crystal Clear: A Human-Powered Methodology for Small Teams;* Addison-Wesley, Reading, Massachusetts, Erscheint voraussichtlich 2002. Online-Version siehe http://members.aol.com/humansandt/crystal/clear

[JBR99] Ivar Jacobson, Grady Booch, James Rumbaugh: *The Unified Software Development Process*; Addison-Wesley, Reading, Massachusetts, 1999; ISBN 0-201-57169-2

[You97] Edward Yourdon: *Death March - The Complete Software Developer's Guide to Surviving Mission Impossible Projects;* Prentice Hall, Eaglewood Cliffs, New Jersey, 1997; ISBN 0-13-748310-4