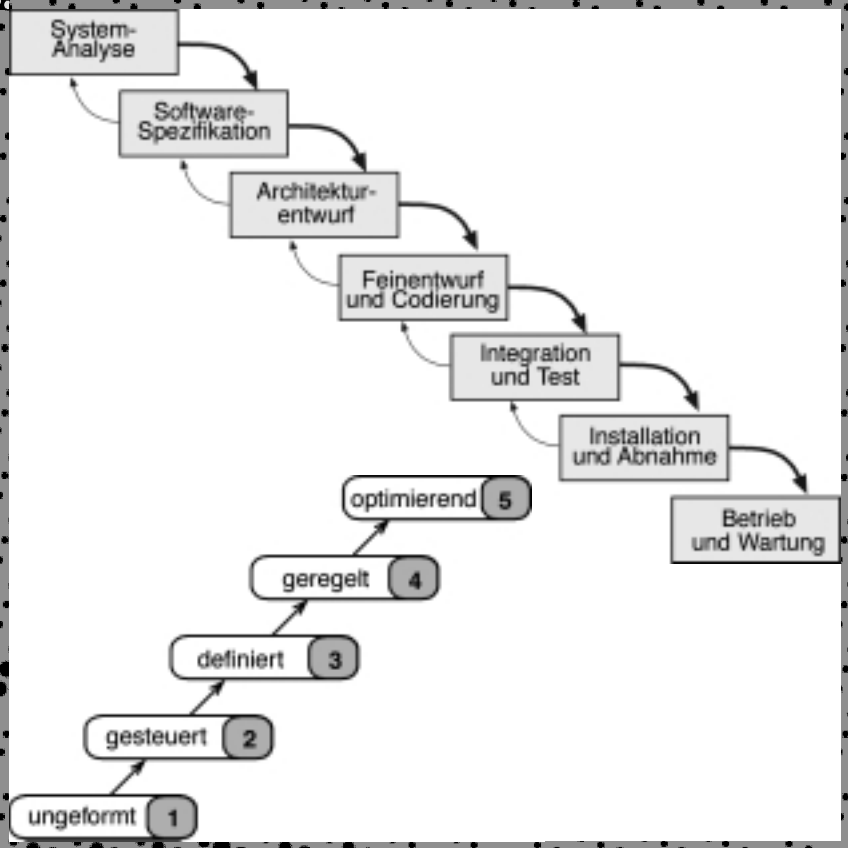
Software-Prozesse und Software-Qualität



Bei einem Motorrad wird diese Präzision nicht aus irgendwelchen romantischen oder perfektionistischen Gründen gepflegt. Es ist nur so, dass die enormen Kräfte von Hitze und Explosionsdruck im Innern des Motors nur mit der Art von Präzision zu bändigen sind, die solche Messgeräte ermöglichen. Bei jeder Zündung treibt die Explosion eine Pleuelstange mit einem Oberflächendruck von mehreren Tonnen pro Quadratzentimeter auf die Kurbelwelle. Wenn der Sitz der Pleuelstange auf der Kurbelwelle präzise ist, wird die Explosionskraft gleichmäßig übertragen, und das Metall hält dem Druck stand. Ist jedoch zwischen Pleuelstange und Kurbelwelle auch nur ein paar hundertstel Millimeter Spiel, kommt der Schub plötzlich wie ein Hammerschlag, und das Lager und die Oberfläche der Kurbelwelle verschleißen in kürzester Zeit, was dann zu einem Geräusch führt, das sich anfangs ganz ähnlich wie ein Ventilklingeln anhört. (aus Pirsig, 1974, Kap. 8)

Was ist Qualität, und was ist Software-Qualität?

Qualität ist ein eigenartiger Begriff, vergleichbar mit Gesundheit und Sicherheit: Wer über Gesundheit nachdenkt, ist krank oder von (eigener oder fremder) Krankheit bedroht. Was wir wirklich erleben, ist nicht Gesundheit, sondern ihr Fehlen. Das gilt ebenso für die Qualität (im landläufigen Sinne): Wir sprechen davon, wenn wir sie vermissen. Dabei ist es meist leicht, die Defizite zu beschreiben: Der Schuh drückt, das Bier schmeckt schal, der Regenmantel ist undicht, das Radio kann die Sender nicht sauber trennen. Der Begriff selbst entzieht sich dagegen der Definition. Pirsig hat sich in seinem oben zitierten Kultbuch darüber Gedanken gemacht, und seine Folgerungen sind alles andere als einfach.

Was er über die Qualität von Motorrädern sagt, kann man auf die Software übertragen: Ein Software-Entwickler sorgt für gute Software-Qualität, weil das praktisch ist, weil gute Software die billigste Lösung ist, also nicht, weil das in irgendwelchen Vorschriften gefordert wird oder ihm durch Drill in der Ausbildung anerzogen wurde.

Aber was ist gute Software-Qualität? Zunächst erwarten wir, dass die Anforderungen erfüllt sind, dass also die Software (Programme und Dokumente) leistet, was sie leisten soll (Brauchbarkeit). Das reicht aber nicht aus; da sich die Anforderungen immer wieder ändern, muss es möglich sein, die Änderungen auch umzusetzen. Die Software muss also wartbar sein (Wartbarkeit). Andernfalls erstarrt sie und verliert ihren Nutzen.

Brauchbarkeit und Wartbarkeit bestimmen gemeinsam die Produkt-Qualität. Ein schlecht geführtes und zerstrittenes Orchester kann keine schöne Musik hervorbringen; damit hohe Produkt-Qualität entsteht, muss das Projekt gut organisiert sein. Die Projekt-Qualität ist also eine Voraussetzung der Produkt-Qualität. In Abbildung 1 ist dieser Zusammenhang schematisch dargestellt. Zur Prozessqualität (ganz links) kommen wir später.

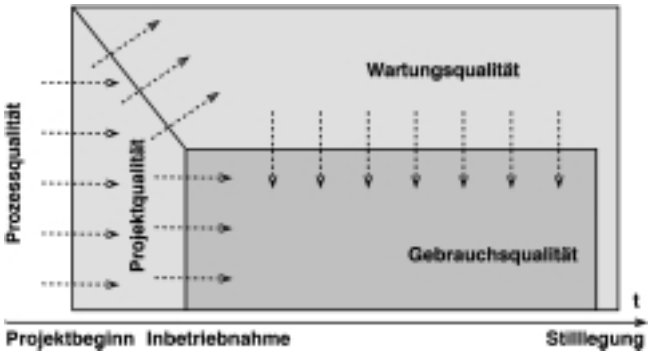


Abb. 1: Qualitätsaspekte und ihr Zusammenhang im Verlauf der Entwicklung und des Betriebs.

Wenn die Software stillgelegt wird, spielt ihre Brauchbarkeit keine Rolle mehr, wohl aber ihre Wartbarkeit, denn Teile der Software, mindestens die Anforderungen, wahrscheinlich auch Testdaten und andere Komponenten, können für das Nachfolgesystem wiederverwendet werden.

Software-Qualitätssicherung

Qualitätssicherung im Sinne vorbeugender Maßnahmen, die für gute Qualität sorgen, gab es in den Hochkulturen seit Jahrtausenden. Die Zünfte des Mittelalters haben die Qualitätssicherung als eine ihrer Hauptaufgaben betrachtet und gepflegt. Dabei standen die wohlorganisierte und geregelte Ausbildung und die geprüfte Qualifikation der zünftigen Handwerker im Vordergrund. Erst zu Beginn des 20. Jahrhunderts wandelte sich das Bild: In der Massenproduktion durch überwiegend ungelernte Arbeiter begann man, die Resultate der Arbeit zu prüfen. Dabei werden die Produkte, also Schrauben oder Rundfunkempfänger oder Autos, mit dem jeweils vorgegebenen Ideal verglichen. Was nicht innerhalb der angegebenen Toleranzen liegt, ist Ausschuss. Das Ideal ist in der Regel nicht materiell vorhanden, sondern durch eine Konstruktionszeichnung, einen Schaltplan mit den geforderten Spannungspegeln und Signalformen oder Vorgaben für die Abgaswerte definiert. Abbildung 2 zeigt ein solches Ideal (eine Zeichnung) und eines der im Allgemeinen sehr vielen Produkte, die dem Ideal entsprechen sollen.

Als in den Sechzigerjahren die Programme größer und komplexer wurden und die Fehler darin immer mehr Probleme machten, lag der Gedanke nahe, die Konzepte der industriellen Qualitätssicherung auf die Software zu übertragen. Das konnte aber nicht gelingen, denn es fehlte das Ideal. Wäre man in der Lage, ein einziges richtiges Exemplar eines Programms zu entwickeln, so könnte man sehr leicht beliebig viele völlig fehlerfreie Kopien dieses Ideals anfertigen. Die Software-Entwickler hatten und haben aber keine solche präzise Vorgabe. Unser Problem besteht nicht in der Massenproduktion vieler Kopien, sondern in der Entwicklung eines einzigen Musters. Es war also eine Qualitätssicherung neuer Art zu entwickeln.

Wir haben bei Software drei Möglichkeiten, die Qualität des Resultats zu fördern:

- Wir können die Arbeit so planen und organisieren, dass die Rahmenbedingungen gute Qualität möglich machen und fördern.
- Wir können bei der Entwicklung solide Arbeit auf dem aktuellen Stand der Technik leisten.
- Wir können die Resultate der Entwicklung prüfen und verbessern oder verwerfen, wenn sie den Anforderungen nicht genügen.

Beschreiben wir die Entstehung der statistischen Qualitätssicherung zu Beginn des 20. Jahrhunderts mit den Begriffen in Abbildung 3, so können wir sagen: Die konstruktive Qualitätssicherung ließ sich mit ungelernten Kräften nicht durchhalten, darum wurde die analytische Qualitätssicherung verstärkt und institutionalisiert. Das ist aber nur ein Behelf, denn mit der Erkennung eines Mangels ist er nicht behoben; liegt der Mangel in der Struktur, so ist es praktisch unmöglich, ihn zu beseitigen. Zudem ist die Erkennung eines Mangels um so schwieriger, je komplexer das betrachtete Objekt ist. Ein Radio oder ein Auto hat so viele Betriebszustände, dass selbst eine sehr aufwändige Prüfung keinesfalls alle Zustände berücksichtigen kann. Software ist das komplexeste Artefakt, das die Menschheit bislang hervorgebracht hat. Darum gibt es kein Verfahren, das die Erkennung aller Mängel wahrscheinlich macht oder sogar garantiert. Wer dagegen die Arbeit gleich richtig macht und sich dessen auch sicher ist, spart die Prüfungen, die Nachbesserungen und vor allem den Ärger mit Fehlern, die nicht erkannt wurden.

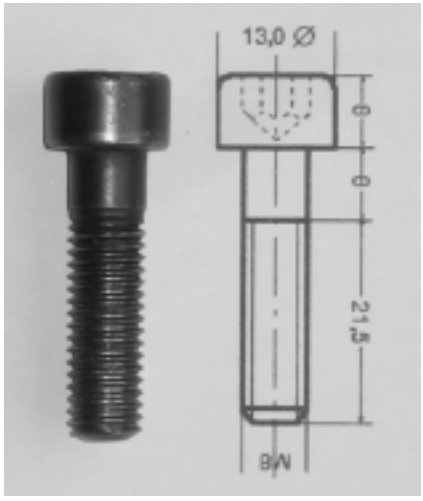


Abb. 2: Ideal und Serienprodukt – wie es bei der Software nicht ist.

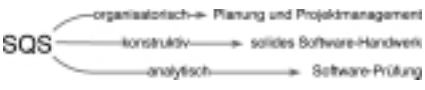


Abb. 3: Elemente der Software-Qualitätssicherung.

Die Entdeckung der Prozesse

Wenn jemand atemberaubend schön Klavier spielt, können wir zwar Begriffe wie „Technik", „Anschlag" oder „Ausdruck" verwenden, um sein Spiel zu charakterisieren. Aber wir wissen dabei, dass unsere Begeisterung gerade *nicht* durch das Kombinieren einzelner Handlungen hervorgerufen wird, sondern durch das vollständige Verschmelzen aller Aspekte in einem glücklichen Moment. Ähnliches lässt sich von allen großen Kunstwerken sagen: Hervorragende handwerkliche Fähigkeiten, also die Beherrschung des Kontrapunkts, die sichere Gliederung einer Leinwand oder die souveräne Bearbeitung eines Marmorblocks, sind Voraussetzung des Kunstwerks, können aber seine Entstehung nicht garantieren.

In den beiden ersten Jahrzehnten der Informatik wurde die Programmierung als Kunst betrachtet, das Programm als Kunstwerk. Das hatte Konsequenzen, vor allem für die beteiligten „Künstler", die Programmierer. Sie nahmen entsprechende Privilegien für sich in Anspruch: Völlige Freiheit bei der Gestaltung, Abneigung gegen Normen, Unzuverlässigkeit bei der Einhaltung der Termine und der Vorgaben für den Aufwand, subjektive Beurteilung der Qualität. Dafür akzeptierten sie auch Nachtarbeit und einen Alltag unter Neonleuchten zwischen Stellwänden in früheren Lagerhallen.

Aber was bei kleinen Programmen noch dann und wann funktionierte, führte bei großen Systemen zum Chaos. Wenn mehrere oder viele Menschen ein Programm entwickeln oder bearbeiten, sind Planung, Systematik und Qualitätssicherung notwendig. Wenn die Anforderungen an ein Programm umfangreich und kompliziert sind, müssen sie dokumentiert und geprüft werden; das Gleiche gilt für andere Informationen, vor allem für den Entwurf der Software und für die Planung des Projekts.

Die Probleme, die durch den traditionellen Ansatz entstanden, waren Ende der Sechzigerjahre so groß, dass sie nicht mehr geleugnet werden konnten. F. L. Bauer stellte 1968 in einem berühmten Satz fest, dass ein Ingenieur-Ansatz notwendig sei: "*What we need is software engineering*!" (Bauer, 1993). In den folgenden Jahren wurde diese Idee zunächst eng, nämlich technisch ausgelegt. Es entstanden die Begriffe „strukturierte Programmierung" und „information hiding", die sehr großen Einfluss auf Programmierung und Programmiersprachen hatten.

Aber Probleme der Planung, der Durchführung und Kontrolle lassen sich nicht durch technische Maßnahmen lösen. So wurde in den Achtzigerjahren auch der Prozess der Software-Entwicklung und -Bearbeitung zum Sanierungsfall erklärt. Was allzuoft nicht gut lief, sollte definiert und systematisiert werden, damit sichergestellt ist, dass alle notwendigen Schritte in der richtigen Ordnung vollzogen werden.

Gewachsene Prozesse

Jeder, der kocht, verwendet ein Kochrezept; das gilt auch dann, wenn er sich nur ein Spiegelei brät. Das Rezept für ein Spiegelei haben die meisten von uns im Kopf, und wir können es auch den konkreten Bedingungen und Wünschen anpassen, indem wir verschiedene Fette, vielleicht auch Bacon oder Speck verwenden und das Ei kürzer oder länger, einseitig oder auch gewendet braten.

So ist es auch bei der Software-Entwicklung. Jeder, der Software herstellt oder verändert, folgt dabei einem Muster, das er erlernt oder den Kollegen abgeguckt oder selbst entwickelt hat. Er wendet dieses Rezept an und variiert es entsprechend den Randbedingungen des konkreten Projekts. Da die meisten Software-Leute in Gruppen arbeiten, gleichen sich die Arbeitsweisen an, wer neu hinzukommt, wird assimiliert. Gruppen entwickeln auf diese Weise einen bestimmten Arbeitsstil, der recht stabil ist, auch wenn die Personen wechseln.

Dieser gewachsene Prozess ist den Beteiligten kaum bewusst. Er wurde nicht systematisch ausgewählt und nicht planmäßig eingeführt, er wird nicht kontrolliert und verbessert. Er ähnelt damit der „natürlichen" Sprache, die unsystematisch entstanden ist und sich weiterentwickelt, die aber *natürlich* alles andere als natürlich ist. Und so, wie verschiedene Menschen verschiedene Sprachen benutzen, so werden auch unterschiedliche Prozesse gelebt. Die allermeisten Menschen suchen sich ihre Sprache und ihren Prozess nicht aus, sondern sind zufällig hineingeboren worden.

Wenn alle Beteiligten mit dem gewachsenen Prozess zufrieden sind und die Resultate nicht nach einer Änderung verlangen, ist dies die optimale Form der Organisation. Wer ohne Brille gut sehen kann, sollte nicht seine Zeit und sein Geld damit vergeuden, die Parameter seiner Augen bestimmen zu lassen, Gläser zu wählen und ein Gestell auszusuchen, um dann ständig nach seiner Brille zu suchen. Die einfachste und beste Lösung ist immer, kein Problem zu haben, das man lösen muss.

Geplante Prozesse

Leider sind die an einem Software-Projekt Beteiligten sehr oft *nicht* mit dem Prozess oder seinen Resultaten zufrieden. Die folgenden Beispiele zeigen, welcher Art die Probleme sind:

- Informationen ungeklärter Herkunft und Qualität werden verwendet.
- Wahlmöglichkeiten werden nicht erwogen.
- Lösungsideen werden ohne gründliche Prüfung ausgewählt oder verworfen.
- Pläne werden aufgestellt und Entwicklungskosten geschätzt, ohne dass die verfügbaren Informationen ausgewertet wurden.
- Zwischenergebnisse, zum Beispiel die Antworten bei der Erhebung der Anforderungen oder die Resultate der Tests, werden gar nicht dokumentiert oder nicht systematisch abgelegt. Darum sind sie später nicht verfügbar.
- Kritische Entscheidungen kommen zufällig zu Stande (beispielsweise in einem Telefongespräch zwischen einem Entwickler und einem Anwender auf Kundenseite) und werden weder geprüft noch dokumentiert.

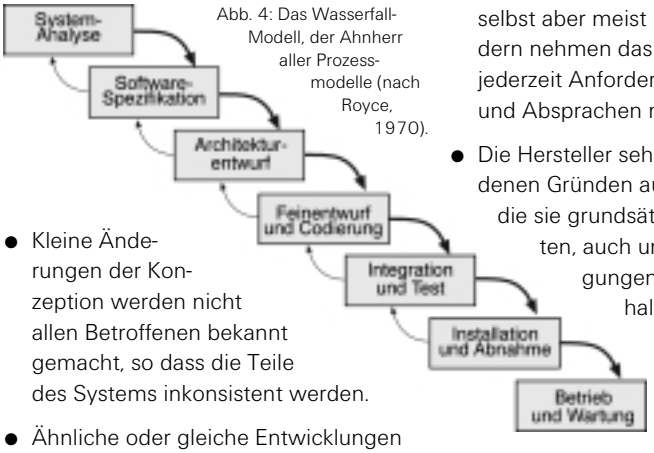


Abb. 4: Das Wasserfall-Modell, der Ahnherr aller Prozessmodelle (nach Royce, 1970).

- Kleine Änderungen der Konzeption werden nicht allen Betroffenen bekannt gemacht, so dass die Teile des Systems inkonsistent werden.
- Ähnliche oder gleiche Entwicklungen (wie die Realisierung bestimmter Programmfunktionen) finden mehrfach statt, weil die Entwickler in einem großen Projekt wenig voneinander wissen.

Solche Mängel der Organisation und Zusammenarbeit haben zur Folge, dass die Bearbeitung der Software mehr Aufwand erfordert und länger dauert, als nötig wäre und dass die Resultate nicht so gut sind, wie sie sein sollten und könnten. Eine naheliegende Konsequenz ist die Vorgabe der Bearbeitungsschritte, also die Definition eines Prozesses, der sicherstellt, dass die oben genannten Probleme vermieden werden. Abbildung 4 zeigt das erste bekannte Vorgehensmodell, das wegen seiner Kaskaden-Struktur als Wasserfall-Modell bezeichnet wird.

Prozessmodelle

Man könnte sich also vorstellen, dass in allen Firmen und Organisationen anerkannte Fachleute ausgewählt wurden und den Auftrag erhielten, einen Prozess zu definieren, der einen akzeptablen Kompromiss zwischen den Interessen aller Beteiligten garantiert, nämlich

- den Interessen der Kunden, die die Software brauchen und die Entwicklung bezahlen,
- den Interessen der Hersteller, die die Projekte akquirieren, die Verträge abschließen und die Entwickler beschäftigen,
- den Interessen der Software-Leute, die die Entwicklung praktisch durchführen.

Tatsächlich ist das aber eine extrem schwierige und meist sehr undankbare Arbeit. Gründe gibt es bei allen drei beteiligten Parteien:

- Die Kunden stellen zwar gern Forderungen an die Hersteller, wollen sich selbst aber meist nicht festlegen, sondern nehmen das Recht in Anspruch, jederzeit Anforderungen zu ändern und Absprachen neu zu interpretieren.
- Die Hersteller sehen sich aus verschiedenen Gründen außer Stande, Regeln, die sie grundsätzlich für richtig halten, auch unter den Randbedingungen der Projekte einzuhalten. Zu diesen Randbedingungen gehören vor allem die Zeitnot, die Instabilität der Anforderungen und die Schwächen der Entwickler (mangelnde Qualifikation und Disziplin).
- Die Entwickler verteidigen ihre Freiheiten, ändern nur sehr ungern ihre Arbeitsweise und haben eine sehr verständliche Abneigung gegen Regeln, die ihnen mehr Arbeit machen, ohne einen erkennbaren Nutzen zu bringen.

Hinzu kommt, dass durchaus kein Konsens darüber besteht, wie ein Software-Projekt ablaufen *sollte*. Wer irgendeinen konkreten Vorschlag macht, hat mindestens die Hälfte, möglicherweise drei Viertel der Leute gegen sich. Kurz: Es gibt einfachere Möglichkeiten, sich mit seinen Kollegen zu entzweien und am Ende vor einem Scherbenhaufen zu stehen.

In dieser Situation ist es vorteilhaft, einen Prozess zu übernehmen, der renommierte Fürsprecher hat, irgendwo anders bereits mit Erfolg eingesetzt wurde und zudem durch Literatur, Web-Seiten, Konferenzen und Seminare gut dokumentiert ist. Damit sind wir bei den *Prozessmodellen* angelangt. Der bekannteste Vertreter dieser Spezies ist der *Unified Software Development Process* oder kurz *Unified Process* (UP) (Jacobson, Booch, Rumbaugh, 1999), der bei Rational (IBM) als *Rational Unified Process* (RUP) vermarktet wird (Kruchten, 2003). Wer darauf zurückgreift, hat sofort eine breite Basis von Lehrbüchern, Werkzeugen und Trainern, die allesamt die Botschaft vermitteln: *So ist es richtig, so sollst du es machen!*

In Deutschland, genauer im Umfeld militärischer Software-Entwicklung, ist ein anderes Modell entstanden, das V-Modell (Dröschel, Heuser und Midderhoff, 1998; Rausch und Niebur, 2005). Auch das V-Modell verspricht dem Anwender Rat und Hilfe bei der Organisation seines Projekts und bei der Suche nach Werkzeugen und „Kochbüchern". Abbildung 5 zeigt die Entscheidungspunkte und den Ablauf eines Systementwicklungsprojekts. Der Tatsache, dass die konkreten Projekte sehr, sehr unterschiedlich sind, wird dadurch Rechnung getragen, dass das Modell generisch ist, also einen Rahmen vorgibt, der vor der Anwendung des Modells ausgeprägt werden kann (und muss). Eine solche Ausprägung könnte grundsätzlich entweder additiv sein (also Leerstellen enthalten, die noch gefüllt werden müssen) oder subtraktiv (also optionale Teile enthalten, die gestrichen werden können). Beim neuen V-Modell XT hat man sich für die subtraktive Variante entschieden, also eine Maximallösung definiert, die von den Anwendern reduziert wird. Allerdings darf man die Schwierigkeit einer solchen Ausprägung nicht unterschätzen. Erfahrungen zeigen, dass keineswegs klar ist, worauf verzichtet werden kann; am Ende bleibt ein Dokumentenkatalog übrig, der vielen Entwicklern noch immer als Albtraum erscheint.

Abb. 5: Projektablauf (aus der Sicht des Auftraggebers) nach dem V-Modell XT.

Prozessbewertung

Das Verteidigungsministerium der USA, das DoD, ist seit langem der mit Abstand größte Software-Kunde der Welt; jedes Jahr werden in neue und veränderte Software Milliarden von US-Dollar investiert. Die Software, die das DoD entwickeln lässt, ist meist Teil eines größeren (Waffen-)Systems, beispielsweise eines Kampfflugzeugs. Wenn die Software nicht rechtzeitig fertig wird oder den Ansprüchen des Auftraggebers nicht genügt, kann das DoD die Bezahlung verweigern; das nützt ihm aber nicht viel, denn nun steht eine teure Hardware zur Verfügung, ein Flugzeug, ein Schiff oder eine Rakete, die ohne Software wertlos ist. Das DoD hat also ein starkes Interesse, dafür zu sorgen, dass bestellte Software mit großer Sicherheit termingerecht und mit den geforderten Eigenschaften geliefert wird.

Aus diesem Grund wurde in den Achtzigerjahren mit Mitteln des DoD eine neue Institution geschaffen, das SEI (Software Engineering Institute). Es wurde am Rande der renommierten Carnegie Mellon University in Pittsburgh, Pennsylvania, angesiedelt und hatte als ersten und wichtigsten Auftrag, eine Art Checkliste zu entwickeln, die es gestattet, Software-Hersteller zu beurteilen. Diese Checkliste wurde als *Capability Maturity Model* bekannt (Jalote, 1999), also als Modell der Prozessreife, die ein Unternehmen nachweisen kann. Das CMM führt zu einer Einstufung auf einer Skala von 1 bis 5 (Abbildung 6), wobei 1 die Eintrittsstufe darstellt. Jede höhere Stufe erfordert bestimmte (zusätzliche) Qualifikationen. Dabei gibt es keine Möglichkeiten, Defizite zu kompensieren; um die Stufe M zu erreichen, müssen alle Kriterien der Stufen 2 bis M erfüllt sein.

Nachdem einige Tausend Einstufungen vorgenommen worden waren, wurde das Modell revidiert und verallgemeinert, so dass mit ähnlichen Kriterien auch die Entwicklungen der Hardware beurteilt werden konnten. Dieses neue Modell, das 2001 vorgestellt wurde, erhielt den Namen CMMI (CMM integrated) (Ahern, Clouse, Turner, 2004; Kneuper, 2006).

Heute ist das CMM/CMMI weit über den militärischen Bereich hinaus ein anerkannter Maßstab, an dem sich Unternehmen messen und messen lassen können. Andere, konkurrierende Modelle sind entstanden, die dem CMM mehr

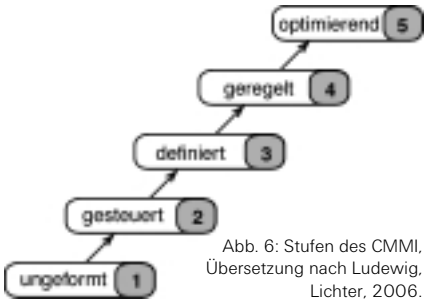


Abb. 6: Stufen des CMMI, Übersetzung nach Ludewig, Lichter, 2006.

oder minder ähnlich sind. Hier ist vor allem SPICE zu nennen (Software Process Improvement Capability dEtermination) (Hörmann, Dittmann, Hindel, Müller, 2006), das in der Automobilindustrie weite Verbreitung findet.

Der Software-Prozess als Reling

Wer ein großes Stück Stahl auf verschiedenen schweren und teuren Maschinen bearbeitet, um es schließlich in ein Produkt einzubauen, hat nicht viele Freiheiten bei der Gestaltung des Arbeitsablaufs; die physische Natur des Werkstücks gibt vor, wie es gelagert, transportiert und verformt werden muss. Ganz anders ist die Situation bei Software. Der Transport erfolgt annähernd mit Lichtgeschwindigkeit; darum können Menschen, die über die ganze Welt verteilt sind, an derselben Software arbeiten. Die Gestaltung der Software ist vor allem durch große Freiheit gekennzeichnet. Weder müssen ganz bestimmte Werkzeuge eingesetzt werden, noch ist von vornherein klar, welche Strukturen sinnvoll und angemessen sind. (Dass die *Veränderung* einer einmal geschaffenen Software sehr schwierig, aufwändig und fehlerträchtig ist, steht auf einem anderen Blatt und wird von vielen Leuten bis heute nicht verstanden.)

Unter diesen Bedingungen ist ein Prozess, der die Entscheidungen der Entwickler sinnvoll einschränkt, ein Segen. Er bietet Führung, so wie die Reling auf einem Schiff dem Matrosen Halt gibt, der sich bei stürmischer See auf dem Deck bewegen muss. Wer Gefahr läuft, in den Gängen des schaukelnden Schiffs zu stürzen oder gar von einem Brecher über Bord gespült zu werden, wird sich nicht darüber beklagen, dass seine Bewegungsfreiheit durch die Reling eingeschränkt ist. Er wird im Gegenteil froh sein, dass er dank der Reling auch in finsterster Nacht den Weg findet.

Ein guter Software-Prozess ist eine solche Reling. Er schafft Sicherheit und Halt. Sollte sich zeigen, dass irgendwo eine Reling im Wege steht, dass der Prozess etwas vorschreibt, was nicht sinnvoll ist, kann und sollte man das ändern. Aber niemand wird auf die Idee kommen, darum alle Relings, alle Regeln für die Projektdurchführung zu beseitigen.

Der Software-Prozess als Prokrustesbett

Die Definition und Durchsetzung bestimmter Prozesse soll die Voraussetzungen verbessern, dass in kurzer Zeit gute, das heißt den Anforderungen entsprechende Software zu möglichst geringen Kosten entsteht. Termineinhaltung und Kosten können fast alle Menschen beurteilen. Aber was ist gute Software? Nur wenige können darauf antworten. Wer Verantwortung für Software-Projekte trägt, aber den großen Zusammenhang nicht überblickt, neigt dazu, formale Kriterien anzuwenden, wenn er die Arbeit seiner Software-Entwickler beurteilt. Damit entsteht eine Situation, die man als spezielle Form der Bürokratie charakterisieren kann, die Prozessokratie: Richtig ist nicht, was gute Software schafft, sondern was den Vorschriften des Prozesses entspricht. Der Prozess ist das höchste Gut, ihm sollen alle dienen.

Um im Bild mit der Reling zu bleiben: Man kann den Matrosen vorschreiben, dass sie immer und unter allen Umständen an der Reling entlang laufen. Aber wenn das dazu führt, dass sie auch bei ruhigem Wetter und Sonnenschein auf der Backbordseite vom Heck zum Bug und auf der Steuerbordseite wieder zum Heck gehen müssen, um von Backbord nach Steuerbord zu kommen, wird das auf wenig Verständnis stoßen. Solche Regeln sind nur noch Selbstzweck, sie haben keinen Sinn.

Vor allem in den USA scheint diese bürokratische Form der Prozess-Anwendung sehr populär zu sein. Möglicherweise spielt dabei die klare Trennung zwischen Management und Technik ein Rolle; in Europa sind wir daran gewöhnt, dass Führung auch fachlich legitimiert sein muss. Aber es scheint, dass diese Sicht auch hier an Einfluss verliert.

So gab es Versuche, den Prozess nicht nur zu definieren, sondern explizit auszuprogrammieren, so dass der Entwickler auf seinem Bildschirm morgens detaillierte Anweisungen vorfindet, was er zu tun hat. Seine Arbeit wird damit der Arbeit am Fließband immer ähnlicher: Mache den Handgriff, für den du eingestellt wurdest, mache ihn schnell und präzise, und denke nicht darüber nach, was die Leute vor dir und hinter dir am Band tun. Aber was am Fließband funktioniert, ist für die Arbeit des Software-Entwicklers Gift; er fühlt sich der Freiheit beraubt, die ihn an diesem Beruf ursprünglich gereizt hatte. Sein Potenzial, bessere Lösungen, Abkürzungen und elegante Verbesserungen zu entdecken und zu realisieren, wird geknebelt. Das macht ihn unzufrieden und zu einem Feind des Prozesses. Der Prozess erscheint als ein Prokrustes-Bett: Der Riese Prokrustes hatte der Sage nach die unangenehme Gewohnheit, seine Gäste in ein Bett zu stecken, das ihnen, wenn sie klein waren, viel zu lang, wenn sie groß waren, viel zu kurz war. Durch das gewaltsame Anpassen, also das Zerreißen der Kleinen und das Abschneiden der Großen, wurden die Gäste dann dem Bett angepasst.

Nur vor diesem Hintergrund ist die Gegenbewegung zu verstehen, die in den letzten Jahren die Diskussion über Software-Prozesse wesentlich geprägt hat: Den, wie die Protagonisten sagen, „schweren" Prozessen wurden die leichten („agilen") gegenübergestellt, die versprechen, frei von allem unsinnigen Ballast zu sein. Abbildung 7 gibt das „Agile Manifesto" wieder.

„Bei uns müsst ihr nicht bürokratischen Vorgaben dienen, sondern dürft tun, was sinnvoll ist und das Software-Projekt voranbringt." Wer in einer Prozessokratie arbeitet, wird diese Botschaft als süßen Gesang der Sirenen hören.

Das bekannteste Modell aus der Gruppe der agilen Prozesse ist das Extreme Programming (XP) (Beck, 1999). Wer es näher betrachtet, stellt verblüfft fest, dass XP den Entwicklern keineswegs freistellt, wie sie arbeiten. Auch XP regelt, teilweise recht genau, wie Software entwickelt werden soll. Das geht bis hin zur Aussage, die „Sitzungen" sollten im Stehen stattfinden, damit die Dynamik der Teilnehmer nicht durch Herumsitzen gebremst wird. Insgesamt bedeutet XP, dass eine Reihe von Regeln, die im Laufe der letzten dreißig Jahre entstanden sind, über Bord geworfen werden. So entsteht weder eine Spezifikation noch eine umfassende Planung für das Projekt. An Stelle einer Spezifikation gibt es den Kunden, der physisch anwesend sein soll und damit als lebende Spezifikation verfügbar ist.

Es gibt Aufgaben (beispielsweise die Entwicklung eines Web-Portals), für die dieses Vorgehen große Vorteile hat. Es gibt aber viele andere Aufgaben, insbesondere solche mit hohen Sicherheitsanforderungen oder mit einer großen, heterogenen Klientel, für die XP ganz sicher nicht geeignet ist.

Ausblick

Wer die Entwicklung im Software Engineering über einen längeren Zeitraum verfolgt hat, kommt nicht umhin, der Feststellung von Fred Brooks (1987) zuzustimmen: *„No silver bullet!"* Die Wunderlösung, das Allheilmittel für alle Probleme der Software-Entwicklung, gibt es nicht und wird es nie geben. Wir entwickeln Systeme von ungeheurer Komplexität; das können die meisten Laien nicht erkennen, weil diese Systeme am Ende in fingernagelgroßen Speichern stecken und wenig kosten. Die Systeme sind aber von großer, in vielen Anwendungen lebenswichtiger Bedeutung. Den meisten Menschen fallen Kernkraftwerke und Herzschrittmacher ein, wenn sie über die Gefahren durch Software nachdenken. Dabei übersehen sie, dass praktisch jegliche Infrastruktur, die wir heute in Anspruch nehmen, von der Wasserversorgung über das Telefon, die Verkehrsmittel, die Banken und Behörden bis zu allen Sicherheitssystemen, im Kern Software enthalten und ohne diese Software nicht arbeiten können. Dass es auch ein Leben ohne Software gibt, sehen wir in manchen fernen Ländern. Wir sehen dort aber auch, dass es mit unserem kaum Ähnlichkeit hat. Wenn wir so leben wollen, wie wir leben, sind wir auf gute, zuverlässige Software angewiesen. Darum bleibt die Qualität der Software ein zentrales Thema der Informatik.

Den einen, allgemein gültigen Prozess wird man nicht entdecken. Es ist gut, dass die Prozessmodelle miteinander konkurrieren; nur so kann es Verbesserungen geben. Die Manager in der Industrie, in den Banken und Verwaltungen wären gut beraten, die gewachsenen Prozesse ernst zu nehmen und notwendige Veränderungen zielstrebig, aber behutsam durchzuführen.

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions	over	processes and tools
Working software	over	comprehensive documentation
Customer collaboration	over	contract negotiation
Responding to change	over	following a plan

that is, while there is value in the items on the right, we value the items on the left more.

Abb. 7: Das „Agile Manifesto" von 2001, Quelle: www.agilemanifesto.org

Literatur/Quellen

Die meisten der Abbildungen in diesem Artikel sind dem Buch von Ludewig und Lichter (2006) entnommen. Über die Literaturangaben darin findet man weitere Referenzen, auch auf Informationen im Internet.

Ahern, D.M., A. Clouse, R. Turner (2004): CMMI Distilled – A Practical Introduction to Integrated Process Improvement. 2nd ed., Addison-Wesley, Boston, MA

Bauer, F.L. (1993): Software Engineering – wie es begann. Informatik-Spektrum, Band 16 (5), 259-260

Beck, K. (1999): Extreme Programming Explained – Embrace Change. AddisonWesley, Boston, MA. Zweite Auflage von 2004 mit Koautorin Cynthia Andres

Brooks, F.P. (1987): No silver bullet – essence and accidents of software engineering. IEEE Computer, Vol. 20 (4), 10-19

Dröschel, W., W. Heuser, R. Midderhoff (1998): Inkrementelle und objektorientierte Vorgehensweisen mit dem V-Modell 97. Oldenbourg-Verlag, München

Hörmann, K., L. Dittmann, B. Hindel, M. Müller (2006): SPICE in der Praxis: Interpretationshilfe für Anwender und Assessoren. dpunkt.verlag, Heidelberg

Humprey, W.S. (1989): Managing the Software Process. Addison-Wesley, Reading, MA

Jacobson, I., G. Booch, J. Rumbaugh (1999): The Unified Software Development Process. Addison-Wesley, Boston, MA

Kneuper, R. (2006): CMMI – Verbesserung von Softwareprozessen mit dem Capability Maturity Model. 2. Aufl., dpunkt.verlag, Heidelberg

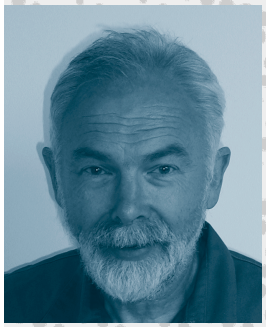
Kruchten, P. (2003): The Rational Unified Process – An Introduction. 3rd ed., Addison-Wesley Longman, Boston, MA

Ludewig, J., H. Lichter: Software Engineering – Grundlagen, Prozesse, Techniken, Werkzeuge. dpunkt.verlag, Heidelberg, 2006

Pirsig, R.M. (1974): Robert M. Pirsig: Zen und die Kunst ein Motorrad zu warten. Aus dem Amerikanischen von Rudolf Hermstein. Fischer Verlag, Frankfurt am Main, 1978. Originalausgabe: Zen and the Art of Motorcycle Maintenance – an inquiry into values. William Morrow & Company, New York, NY, 1974

Rausch, A., D. Niebuhr (2005): Erfolgreiche IT-Projekte mit dem V-Modell XT. OBJEKTspektrum 3/2005 (Mai 2005), 42-50, Rausch, Niebuhr, 2005

Royce, W.W. (1970): Managing the development of large software systems. IEEE WESCON, Los Angeles, CA, 1-9; nachgedruckt in Proceedings of 9th ICSE, Monterey, CA, IEEE Computer Society Press, 328-338



Prof. Dr. Jochen Ludewig

Jahrgang 1947, Studium der Elektrotechnik an der TU Hannover und der Informatik an der TU München; nach fünf Jahren im Kernforschungszentrum Karlsruhe (Promotion) im BBC-Forschungszentrum in Baden/Schweiz als Gruppenleiter Software Engineering (bis 1985), Assistenzprofessor an der ETH Zürich, seit 1988 auf dem Lehrstuhl Software Engineering der Universität Stuttgart. Schwerpunkte Softwareprojekt-Management, Software-Qualitätssicherung, Software-Wartung, Metriken, Didaktik des Software Engineerings. Entwurf und Einrichtung des Diplomstudiengangs Softwaretechnik seit 1996. Autor einiger Lehrbücher und Brückenbauer zwischen Hochschule und Industrie.