

Emergente Software - Entwicklungsprozesse (*emergent software development processes*)

Version 2.22, © Juni - Dezember 2006, Guido Stepken

„Man kann am Ende der Entwicklung keine Qualität in ein System hineintesten!“ - Dieser Beitrag beschreibt neuartige, aus der Kybernetik (Steuermannslehre) stammenden Lösungsansätze derjenigen Probleme, die bei mittleren und großen Software - Projekten (C, C++, JAVA, .NET) typischerweise auftreten („systemisches Management von IT-Projekten, Risiko-Management, Kostenmanagement, Komplexitätsmanagement“). Gewünscht ist stets die schnelle, zeitgenaue und preiswerte Lösung. Komplexe Software ist nicht nur hochgradig fehlerbehaftet, sondern auch in der Herstellung oft viel teurer, als kalkuliert, immer hinter dem Zeitplan zurückliegend. Stundenzettel, MS-Project, Groupware mit Todo-Listen, Zielvereinbarungssysteme, Motivations-Kurse, NLP, Projektmanagement, streng hierarchisch nach dem „Harzburger Modell“ geordnet, mit Kontrollstrukturen alleorts. Prinzipienreiterei, unausgereifte Methoden, ... führen nicht nur dazu, daß die Zahl der Häuptlinge in Relation zu den Indianern ansteigt, oder kurz vor Scheitern des Projektes zusätzliche Programmierer engagiert werden, die die anderen nur von der Arbeit abhalten, sondern auch zu mangelhafter Software, welche darüber hinaus noch kaum zu warten ist. Umso erstaunlicher sind die Erfolge der freien Softwareentwicklung, wo eine wilde Horde von Hackern ohne erkennbare Planung und kaum erkennbarer Organisation es schafft, Softwareprojekte mit vielen Millionen Zeilen fertigzustellen, zu warten und weiterzuentwickeln, siehe z.B. <http://www.little-idiot.de/teambuilding/apacheentwicklung.gif> oder z.B. bei dem Linux Projekt?

Wie gelingt es in freien Projekten, ohne autorisierte Projektleitung, ohne Projektmanagement, ohne hierarchische Strukturen, Entscheidungen zu treffen, Aufgaben zu verteilen und zu koordinieren, qualitativ hochwertige Software zu produzieren, die darüber hinaus noch leicht zu warten ist?

Keine Kaffeekränzchen, keine dauernden Besprechungen, keine kontraproduktiven Bedenkenträger: „*Habe ich schon immer gesagt...*“, kaum innere Reibung.

Teams mit emergenten Strukturen (wo das Team mehr ist, als die Summe seiner Teile) zeigen **keine „Leistungsorientierung“** mehr, sondern sind „Wirkungsorientiert“, insgesamt viel produktiver, kommunizieren intensiver und häufiger, hören einander besser zu, akzeptierten eher Lösungsvorschläge von anderen, bringen bessere Lösung hervor, zeigen mehr Hilfsbereitschaft und Freundlichkeit, entwickeln grösseres Vertrauen in die eigene Gruppe, weisen eine grössere Arbeitsteilung auf, haben ihre Aktivitäten effektiver koordiniert

Das Geheimnis liegt in den Software – Produktions – Prozessen selber, welche automatisch Qualität im Code und Wartbarkeit hervorbringen, ohne daß mit aufwändiger Qualitätssicherung, Vorschriften, Regelwerken und Testverfahren nachgeholfen werden muß: **„Man kann am Ende der Entwicklung keine Qualität in ein System hineintesten!“** Der Trick dabei ist immer derselbe – Sämtliche Prozesse sind als „Regelkreise“ aufzufassen, welche ineinandergreifen und rückgekoppelt sind. Kein Prozess, der nicht immer wieder durchlaufen wird, und dabei sowohl für ständige Verbesserung der Qualität der Software sorgt, indem Nachfolger die Fehler des Vorgängers entdecken und beheben, sondern auch Wissen um Programmieretechniken im Kreis weitergereicht werden, sodaß ein Programmierer vom anderen lernt, und schließlich alle sich auf einem sehr hohen Niveau bewegen. „Abfallprodukt“ dieser Prozessgestaltung ist, daß alle Mitarbeiter sogar redundant sind, sich gegenseitig bei Urlaub und Krankheit vertreten können.

Welche impliziten Logiken sich hinter den verschiedenen Softwareentwicklungsmodellen, also den prozessualen Abläufen verbergen, und wie diese mit den psychologisch / menschlichen Verhaltensweisen wechselwirken, sei hier anhand einiger Beispiele aus der Praxis beleuchtet. „Emergente Prozesse“, bzw. „**Psychodynamisches Prozessdesign**“ sind das Geheimnis hinter den ausgefeilten Softwareentwicklungsmodellen. Der Begriff „Emergenz“ stammt aus der Kybernetik und besagt folgendes: „Das Ganze ist mehr, als die Summe seiner Komponenten!“. Hierunter versteht man nicht die typischen Synergie-Effekte, welche durch den Einsatz von einem Entwicklungsframework oder Rapid Development (RAD) – Toolkits, bzw. Model Driven Architecture (MDA) entstehen, sondern **unter Emergenz versteht man vor allem scheinbar aus dem Nichts entstehende, neue Möglichkeiten** der effizienteren Zusammenarbeit im Team, der einfacheren Verteilung (Komplexitätsreduktion) von Programmier(teil)aufgaben und somit der Vereinfachung der Programmstrukturen, die übrigens sehr oft erstaunliche Parallelen zu den Kommunikationsstrukturen im Team aufweisen. Emergente Prozessketten können die Komplexität der Zusammenarbeit im Team erheblich reduzieren, bzw. dorthin verschieben, wo sie nicht mehr auffällt, was auch ganz erhebliche Auswirkungen auf „Test Driven Development“ (TDD), also Qualitätstests hat. Der folgende Beitrag beleuchtet moderne Softwareentwicklungsmodelle aus dem Blickwinkel der Emergenz.

Zunächst ein Beispiel für Emergenz. Jeder hat schon einmal das HTTPS – Protokoll verwendet, bei welchem eine verschlüsselte Verbindung zwischen Browser und Server (z.B. für Homebanking) zustande kommt. Da fragt man sich doch gleich, wo die Passworte für Ver- und Entschlüsselung ausgetauscht worden sind, ohne daß jemand Drittes den Datenverkehr mitlesen kann. Die Antwort ist: Es wurden niemals Passworte ausgetauscht, dennoch kommt ein verschlüsselter Krypto-Tunnel zustande. Wie kann das? Die Denkstrukturen von Otto-Normalbürger sind hier sicher überfordert. Die komplexen Lösungen der Zero-Knowledge-Probleme bieten hier Ansätze, genauer gesagt, richtige „Trickkisten“ an:

Man stelle sich eine Kiste vor, welche mit zwei Vorhängeschlössern verschließbar ist. Nun lege man eine Nachricht in die Kiste, schließe mit seinem eigenen Vorhängeschloß ab, den Schlüssel behalte man für sich. Nun versendet man diese Kiste per Post. Niemand Drittes kann den Inhalt lesen, da diese ja verschlossen ist. Nun bittet man den Empfänger, sein eigenes Schloß als zweites Schloß anzubringen, den Schlüssel für sich zu behalten, und die Kiste zurückzusenden. Die Kiste wieder in Händen, entfernt man das eigene Schloß, und sendet die Kiste abermals zurück. Der Empfänger kann nun sein eigenes Schloß entfernen, den Inhalt lesen. Zu keinem Zeitpunkt wurde ein Schlüssel ausgetauscht, und zu keinem Zeitpunkt war die Kiste unverschlossen unterwegs. Die Schlüssel, stellvertretend für Passworte oder Keys wurden nicht übermittelt. Durch den emergenten Prozeß des dreimaligen Hin- und Herschickens wurde die Übermittlung eines Schlüssels überflüssig. Und wenn man nun sich das PGP – Verfahren anschaut, welches auf diesem emergenten Prinzip aufbaut, so steht hier stellvertretend für die doppelt verschließbare Kiste z.B. die Primfaktorzerlegung eines zwei – bzw. dreifachen Produktes von langen Primzahlen.

Wo nun liegt die Emergenz dieses Prozesses? Dieser Prozess erst ermöglicht die täglichen, milliardenfachen, sicheren Datenverbindungen ohne riesigen Verwaltungsaufwand, wo Passworte oder Schlüssel auf verschiedenen Wegen übermittelt werden müssen, spart also immenses Geld ein. Dieser emergente Prozess ist unter dem Namen „Diffie Hellmann Key Exchange“ bekannt, jedoch ist das Prinzip allgemein kaum wirklich verstanden, zumindest nicht als „emergenter Prozeß“. Hier ist Deutschland noch eher Entwicklungsland.

Der Begriff Emergenz beschreibt also Prozesse, welche durch ihre impliziten Logiken nachfolgende Prozesse erst ermöglichen, welche daraufhin alt eingefahrene Prozesse ablösen können, also Herstellung, Produktion erheblich beschleunigen oder Ressourcen sparen.

Eigentlich typisch für Emergenz sind rekursive, immer wieder erneut startende Prozesse, bei welchem das Resultat eines Prozesses eine verändernde, rückkoppelnde Wirkung hat. Der Mensch, als „eigenständiger Agent“, erkennt die Auswirkungen seinen Handels, verändert die Anfangsparameter, Strukturen, sogar die prozessualen Abläufe selber, und startet den Prozess aufs Neue. Dazu später mehr.

Ein weiteres Beispiel betrifft die Leistungskontrolle, die leider nicht unabhängig vom Entwicklungsprozess selber sind. Wie messe ich die Arbeitsleistung eines Programmierers mit möglichst geringem Zusatzaufwand? Zahlreiche Softwarehersteller haben hier unterschiedliche Verfahren, welche allesamt einen riesigen Verwaltungsaufwand erfordern, also das Produkt enorm verteuern. „*lines of code written*“ ist ganz sicher kein Maßstab, an welchem man einen Programmierer messen könnte. Es gibt für einen Programmierer immer Möglichkeiten, die Codebasis aufzublähen, durch Cut-und Paste (auch eine Art von Code-Reuse !-), wobei 3000 Zeilen auch durch wenige hundert Zeilen hätten ersetzt werden können. Aufgeblähter Quellcode durch Codegeneratoren, z.B. verteuert das Refactoring, sprich die Softwarewartung enorm. Statistiken bei großen Softwareprojekten haben ergeben, daß **ein Programmierer im Laufe von mehreren Jahren täglich nur ca. 5 endgültige, bleibende Codezeilen zum Produkt beiträgt**. Ständige Strukturänderungen, das oft notwendige Verwerfen von Modulen sorgen dafür, daß die Arbeitsleistung enorm sinkt. Kriterium sollte also vielmehr sein: „lines of software not written“ - die Abgrenzung zum Nichtstun fällt hier schwer....

Ein gutes Design – Pattern, entworfen von einem erfahrenen Software-Architekten, so die allgemeine Denkweise, Sorge dafür, daß im Laufe der Entwicklung nur wenig Code verworfen werden muß. Gute Design-Pattern sorgen gewöhnlich dafür, daß die Kosten der Softwareentwicklung auf 1/3 bis 1/4 reduziert werden können. So effizient nämlich war das Nachprogrammieren von SUN's J2EE. Die Programmierer des Clones JBOSS hatten schließlich von J2EE ein perfektes Design-Pattern vorgegeben, das Nachprogrammieren der Routinen war nur reine Fleißarbeit, mit minimalem Test- und Koordinationsaufwand. Dasselbe gilt hier auch für den Windows Emulator WINE. Insbesondere Frameworks, allen voran J2EE, .NET, MONO, JBLUE (in Schulen als Lehr-RAD oft verwendet), QT4 (KDE), GTK+ (GNOME) und zahlreiche RAD – Werkzeuge enthalten implizite Design-Patterns, an denen sich Programmierer orientieren können, um schnell produktiv zu sein. Hierbei kann die Arbeitsleistung des Teams durchaus auf durchschnittlich 200-500 Codezeilen je Tag ansteigen, sich also durchaus „verhundertfachen“.

Hierin liegt also ein riesiges Potential für Emergenz, wo ein ausgereiftes Design-Pattern und korrekte Anreizsysteme, welche eine Verhaltensänderung im Team bewirken (PD² = Psychodynamisches Prozessdesign) für **überproportionale Leistungssteigerung des gesamten Programmierer – Teams sorgt**, vornehmlich durch Mitdenken im Team, siehe <http://www.little-idiot.de/teambuilding/SynergienMitdenken.pdf>. Hierbei wird die unvermeidliche innere Reibung, der Verwaltungs-, Test-, Koordinations – und Kommunikationsaufwand stark reduziert und bei weitem überkompensiert.

Was aber, wenn einfach kein bewährtes Design – Pattern vorliegt, wie z.B. bei dem kürzlich eingestellten Bundesprojektes FISCUS? Die Erfahrungen der hierfür engagierten TOP-Software – Architekten (J2EE) führten dazu, daß lange Zeit das Projekt gut voranging, bis sich zu einem recht späten Projektstand herausstellte, daß das Design doch nicht tauglich war.

J2EE mit ECLIPSE/Rational Rose (IBM) funktioniert völlig anders: Jahrelang werden Programmierer eingesetzt, alle prozessualen Abläufe als UML – Schemata zu malen, danach wirft der Codegenerator eine Code/Datenbankstruktur mitsamt Ein- und Ausgabemasken in Form von Millionen Zeilen Code aus, und dann kann die eigentliche Logik implementiert werden. Wenn ein solches Softwareprojekt scheinbar gut vorangeht, ist dies noch lange keine Garantie dafür, daß nicht kurz vor Schluß doch noch wegen unlösbarer Probleme der Großteil des Codes neu entwickelt werden muß. Beim Projekt FISCUS wurden viele hundert Mannjahre letztendlich verschwendet, geschätzt etwas unter einer Milliarde Euro.

Das Software-Design-Pattern von J2EE, die saubere Trennung von Business Logik, GUI und Workflow, die hin- und

herflitzenden Container, die sich mitsamt Daten und zugehörigen Programmen eine CPU im SUN-Netzwerk suchen, stellte sich als Blödsinn heraus. Aber das System skaliert ja laut Angaben von SUN, nur leider nicht mit riesigen Datenbeständen, sondern J2EE ist ausschließlich gedacht für kleine Datenbestände, die zusammen mit kleinen Programmen in Container verpackt durch das Netzwerk rasen, auf der Suche nach einer CPU, die gerade keine Last hat ... Dementsprechend war jedes Softwareentwicklungsmodell unter J2EE zum Scheitern verurteilt. Das Design von J2EE ist nicht für diese Art von Anwendung gedacht gewesen.

Zurück zur Beurteilung der Programmierer – Leistung – diese ist enorm Abhängig von einer Vielzahl von Faktoren - am allerwenigsten haben in Großprojekten die individuellen Fähigkeiten des einzelnen Programmierers damit zu tun, wennauch oft Know-Know Unterschiede innerhalb des Teams bemängelt werden, welche jedoch leicht durch regelmäßiges Pair-Programming beseitigt werden können.

An Software-Metriken fehlt es nicht. Zuse [23], Fenton [8], Dumke [6] und andere weisen auf mehr als 150 einzelne Metriken hin.

Es gibt Maße für die Software-Größe, wie Lines of Code, Anweisungen, Function-Points, Data-Points, Feature-Points und Object-Points [19]. Es gibt Maße für die Software-Komplexität, z.B. Halstead's Sprachkomplexität [9], McCabes Graphkomplexität [16], Henry's Schnittstellenkomplexität [11] und Chapin's Datenkomplexität [3]. Schließlich gibt es Maße für die Software-Qualität, wie Fehlerraten, Mängelraten, Verständlichkeit, Testbarkeit, Portierbarkeit und Modularität [12]. Es sind auch Maße für die Software-Wartung [20] sowie für die Software-Wiederverwendbarkeit [17] vorgeschlagen worden.

Für die Schätzung der Wartbarkeit ist der Code Maintainability Index von Welker, Oman und Atkinson besonders empfehlenswert [22]. Für die Bewertung der Wiederverwendbarkeit haben Cimitile, De Lucia und Minro wichtige Beiträge geleistet [4].

Die empirische Untersuchung von Software-Reuse in der Wartung von Li hat gezeigt, wie man wiederverwendbare Funktionen erkennen kann [15].

Diese Metriken beziehen sich jedoch nur auf die Stufe der elementaren Bausteine, der Prozeduren. Hier leisten sie gute Hilfe bei der Detailauswahl wiederverwendbarer Code-Abschnitte, helfen aber wenig bei der Entscheidung, ob ein System überhaupt wiederverwendet werden sollte bzw. kann. Wirtschaftsmanager brauchen einfache, leicht zu ermittelnde Meßwerte, die sich in eine Entscheidungsregel leicht einfügen lassen. Nur so können Software-Metriken zur groben Entscheidungsfindung beitragen.

Außerdem muß zwischen der Art der Wiederverwendung unterschieden werden. Es werden andere Meßwerte benötigt, je nach dem ob konvertiert oder gekapselt wird, denn jede Migrationstechnik hat andere Voraussetzungen.

Häufig wird viel zuviel Wert auf diese Test – und Beurteilungsmethoden gelegt, und viel zu wenig auf das „Psychodynamische Prozessdesign“ des Herstellungsprozesses selber, welches Programmierern keine andere Möglichkeit läßt, außer perfekt fehlerfreien, übersichtlichen, lesbaren und leicht wartbaren Code zu schreiben.

Emergenz - **„Das Ganze ist mehr, als die Summe seiner Komponenten“** ist hier für die Leistung des Teams insgesamt verantwortlich. Im Grunde liegt bei den erfahrenen Programmierern eine große Chance – sie können durch Weitergabe ihres Wissens die jungen Kollegen mitziehen und die Leistung des Teams erheblich steigern. Eine einzige gute Idee, z.B. kann dem Team insgesamt enorm Entwicklungszeit einsparen, wie z.B. die Erkenntnis, **daß komplexe Datenstrukturen und einfacher Code effizienter sind (auch in der späteren Wartung des Codes), als einfache Datenstrukturen und komplexer Code**. Warum angesichts dieser Tatsache immer noch Entscheider haufenweise auf SQL – Datenbanken setzen, und nicht auf OO-Datenbanken bzw. Postrelationale Datenbanken, allen voran PostgreSQL (dessen Kern ist OO!), GOODS, ZODB oder die kommerzielle Datenbank CACHE von Intersystems, ist mir ein Rätsel. Persistenz, hyperdimensionale Datenbankstrukturen, einfaches Datenbankdesign, erheblich höhere Leistung und Verfügbarkeit gegenüber den SQL – Marktführern Oracle, MS SQL, ... sowie stark vereinfachte Programmierung sprechen klar für OO-Datenbanken. SQL Datenbanken beherrschen leider nur einfache Datenstrukturen, die Komplexität des Programmiercodes verkompliziert sich daher enorm. Damit steigen auch die langfristigen Wartungskosten auf ein Vielfaches. IBM hatte schon in den 70'er Jahren ermittelt, daß die Kosten für einen „verschleppten“ Fehler exponential mit der Zeit, die er unentdeckt bleibt, ansteigen.

Kontrolle der Leistung

Die Messung der Leistung eines Programmierers kann also nur anders erfolgen, hierzu möge folgendes Beispiel dienen: Wie kontrolliert ein Manager eines Restaurants seine Angestellten, Kellner, Küche, ohne ständig mit seiner Nase dabei zu sein? Er installiert Kontrollprozesse! Die Kellnerin nimmt eine Bestellung für ein Essen auf, „bongt“ in die Kasse, diese druckt die Bestellung für die Küche aus. Die Küche stellt bereit das Essen zu, die Kellnerin liefert es und kassiert. Der Manager habe z.B. 25 Steaks eingekauft, die Küche 25 zubereitet, die Kellnerin jedoch nur 23 verkauft – hier betrügt die Kellnerin. Hat die Küche nur 23 zubereitet und die Kellnerin 23 verkauft, so betrügt der Koch. Ein einfacher Abgleich von Einkaufslisten, Bestellungen und der Kasseneinnahmen zeigt sofort, ob jemand betrügt, wieviel Trinkgeld gegeben wurde, u.s.w. Dies sind Kontrollprozesse, genauer prozessuale Abläufe, die von dem Management vorgegeben wurden, und welche eine perfekte Kontrolle ergeben, ohne daß der Manager ständig anwesend sein muß.

Eine stichprobenartige Kontrolle der Vorräte schadet natürlich auch nicht, schließlich könnten Kellnerin und Koch sich ja abgesprochen haben, nebenher privat einzukaufen und zu verkaufen.

Die implizite Logik der Kontrolle im Restaurant liegt einfach darin, daß in einem geschlossenen Kreislauf bei den verschiedenen Verarbeitungsprozessen die Stückzahlen unabhängig gemeldet/gebongt werden müssen, und anschließend zu Kontrollzwecken gegeneinander abgeglichen werden können.

Toyota und Porsche haben z.B. bei der Fließband-Produktion eingeführt, daß der nachfolgende Arbeiter die Arbeit des Vorgängers kontrolliert, damit sich die Fehler im Laufe von 100 Arbeitsgängen nicht potenzieren, sondern gegenseitig aufheben. Bevor nicht die Fehler behoben sind, läuft das Band nicht weiter, oder das Auto wird aus dem Band geschubst, nachgebessert, und wieder eingegliedert. Der Druck der Gruppe zwingt jeden Arbeiter, präzise und genau zu arbeiten – hier „fühlen“ die Mitarbeiter eher Verantwortung gegenüber ihren Kollegen, nicht gegenüber dem Vorgesetzten, eine Art sanfter Gruppenzwang zur Präzision hin. Ford und Volkswagen (siehe Geländewagenproduktion in Wolfsburg) haben gerade ein neues Akkord – Modell eingeführt, welches bei Fließbandarbeit nur die Stückzahlen der 100% perfekt gefertigten Autos zählt, OK-Akkord genannt. Hier besteht der Lohn aus einem Grundlohn für den Akkord und Prämien, welche nur gezahlt werden, wenn die Auto's 100% ok sind. Die impliziten Logiken der psychologisch / menschlichen Eigenschaften von Mensch wechselwirken hier mit den prozessualen Abläufen. Kybernetisches Management gibt hier nur die Rahmenbedingungen vor, und sorgt damit dafür, daß sich das Verhalten von tausenden Mitarbeitern („eigenständige Agenten“) im Sinne der Firmenleitung verändert.

Schlechte Manager haben jahrelang jedem Mitarbeiter genauestens vorgeschrieben, welche Arbeiten/Handgriffe wie zu machen sind, moderne Manager gestalten sog. Meta-Prozesse (meta=(griech) über), definieren nur wenige Randbedingungen, genau wissend, daß die Mitarbeiter als sog. „eigenständige Agenten“ mit viel Begeisterung und Engagement, sprich „intrinsische Motivation“ alles notwendige eigenverantwortlich koordinieren werden, um ihr Einkommen zu optimieren.

Ich hatte das Vergnügen, die Firma CDOT, Bangalore, India besuchen zu können, welche ca. 1200 hochqualifizierte Programmierer (ca. 35.000 Menschen sind indirekt für CDOT beschäftigt) beschäftigt. Es gibt quasi kein Problem, welches nicht fertig programmiert in den Schubladen dieser Firma liegen würde. Auch dies ist Emergenz. Die Codebasis muß nur speziell auf die Belange des Kunden angepasst werden. Die Firmenleitung führt sehr erfolgreich dieses Unternehmen mit kybernetischen Modellen, welche Synergien schaffen. Hier ein Ausschnitt:

HR PHILOSOPHY & POLICIES

HRD right from its conception acted as catalyst towards greater synergetic effect so that personnel with individual brilliance and limitations could team up to complement each other for best results.

Ein klares Beispiel für Emergenz kann man auch bei einem sehr verblüffendem Experiment von Reynolds 1987 beobachten, welcher die hochkomplexen Flugbewegungen eines Vogelschwarms beschreibt:

1. Kollisionsvermeidung: Vermeide Kollision mit Nachbarn oder anderen Hindernissen
2. Geschwindigkeitsanpassung: Passe deine Geschwindigkeit der deiner Nachbarn an
3. Schwarmhaltung: Versuche, in der Nähe deiner Nachbarn zu bleiben

Diese 3 einfachen Regeln genügen bereits, um das komplexe Verhalten eines fliegenden Vogelschwarms zu simulieren, welcher fliegend Hindernissen ausweicht, sich in zwei Schwärme aufteilt, um ein größeres Hindernis zu umfliegen, über Berge und durch Täler fliegt. Ein deutscher Beamter oder Angestellter, sprich miserabler Manager (unsere Führungskräfte im Staatsdienst bzw. öffentlichen Dienst wollen einfach nicht dazu lernen) würde wohl eher auf die Idee kommen, jedem einzelnen Vogel entsprechend seinen Fähigkeiten eine genaue Flugbahn vorzuschreiben, und individuelle Verhaltensvorschriften zu machen. Genaugenommen würde jede Biene somit durch die Flugtauglichkeitsprüfung für Piloten fallen.

Ein Kybernetiker, der das Prinzip für Emergenz verstanden hat, schreibt nur drei Regeln für prozessuale Abläufe vor, wohl wissend, daß jeder Vogel als Individuum als „eigenständiger Agent“ selber entscheiden kann, welche Flugbahn dieser einschlägt, und nach welchen Kriterien er dies als „*eigenständig handelnder Agent*“ entscheidet.

Was lernt man hieraus für die Kontrolle von großen Software-Teams? Analysiert man die impliziten Logiken des Restaurant-Beispiels und des Vogelschwarms, so fällt auf, daß die Kontrolle nicht in der genauen Messung der Arbeitsleistung des einzelnen besteht, also die Erfassung der Tätigkeiten eines Programmierers über Stundenzettel, „*lines of code written/not written/changed/deleted/refactored*“, sondern die emergente Leistung des Teams insgesamt muß beurteilt und gemessen werden. In einem Netzwerk von Programmierern als „*eigenständig handelnde Agenten*“ ergibt sich ein engmaschiges Geflecht von Abhängigkeiten der verschiedenen Codemodule untereinander, Programmierer – und Änderungswünsche an die Kollegen, die für die Nachbarmodule zuständig sind. Diese und nur diese können korrekt die Leistung, Einsatzbereitschaft, und Qualität der Arbeit ihres Kollegen korrekt beurteilen. Ein einfacher Beurteilungsfragebogen, der nur abfragt, mit welchem Programmierer oder Gruppe der Programmierer zu tun hatte, wie er die Leistung und Qualität seines Kollegen einschätzt, genügt.

Die impliziten Logiken des Extreme Programming, siehe auch <http://www.little-idiot.de/xp/>, beinhalten eine vergleichbare Art der mündlichen Beurteilung der Leistung der Kollegen, jeder muß im Abstand von 1-2 Wochen regelmäßig die Resultate seiner Arbeit vor dem Team vorstellen, wie weit er gekommen ist, welche Probleme

aufgetreten sind, und die Klärung, warum. Die impliziten Logiken der prozessualen Abläufe von XP führen dazu, daß der Programmierer selber seine Leistung korrekt einschätzen lernt, wodurch die Projektleitung dann erst in der Lage ist, realistisch Kosten und Termine nennen zu können. Der Programmierer mag sich nur ungern die Blöße geben, seine zugesagte Arbeitsleistung wiederholt nicht erreicht zu haben, besonders dann wenn Prämien daran gekoppelt werden. Man beachte hierbei die Rückkopplung – der Mensch führt einen Prozess aus, erhält zeitnahes Feedback, verändert dadurch erst sein Handeln. Liegt zu lange Zeit zwischen Handlung und Feedback, so ist die Wirkung verpufft. Alle emergenten Prozesse sind explizite Rückkopplungen über kurze Wege, rekursive Prozesse mit Feedback. Wichtig ist auch die Art und Weise der Berichterstattung, des Reportings. In der Citybank z.B. werden Berichte grundsätzlich 2-3 Etagen „höher“ abgeliefert, und nicht an den direkten Vorgesetzten, damit halt Mißstände unter keinen Umständen geheim gehalten werden können.

Nicht wenige Führungskräfte versuchen, ihre Programmierer – Teams mit „**Zielvereinbarungssystemen**“ in den Griff zu bekommen. **Ausweichlogiker - „Kann nicht kommen, Lüge kommt später!“** - sorgen dafür, daß Zeitvorgaben nicht eingehalten werden, ganze Ketten von Unterprojekten platzen oder verschoben werden müssen, das Gesamtprojekt aus dem Ruder gerät. **Ausreden allerorts, z.B. „wußte nichts davon“, „habe ich nicht gelesen“, „mich hat niemand informiert“**, u.s.w. sind klare Anzeichen für gewaltige Fehler in der Führungskräfte, die es offensichtlich nicht verstehen, ihre Mitarbeiter zu Offenheit, Geradlinigkeit, Ehrlichkeit, Eigenverantwortlichkeit, Verantwortung für Kollegen und Prozesse, Termintreue, u.s.w. zu erziehen. Die Fehler liegen in der Unfähigkeit, „psychodynamische Prozesse“ korrekt zu entwerfen und in der Unternehmenskultur, also dem „Set“ von geschriebenen und ungeschriebenen Regeln, zu integrieren. Stattdessen versuchen sie, ihre Mitarbeiter über „Zielvereinbarungssysteme“ in ihren Aussagen über Fertigstellungsterminen festzunageln, ohne jedoch zu berücksichtigen, *daß ein Mitarbeiter, der partout nicht will, immer einen Schuldigen findet, welcher für sein eigenes Versagen verantwortlich gemacht werden kann, bzw. eine Situation immer so gestalten kann, daß es nach außen so aussieht, als wären andere dafür verantwortlich.*

Die Installation eines Zielvereinbarungssystems wird in diesem Fall dann auch nichts nützen, weil das, was an „Informationen“ darin abgelegt ist, mit der Wirklichkeit bald nichts mehr gemeinsam haben wird. Die Mitarbeiter werden dann wunderhübsche Planungen recht zeitaufwändig dort „hineinhäckseln“ und dennoch werden Termine reihenweise platzen, aufgrund von „unvorhergesehenen Ereignissen“, wo mal wieder irgendjemand im Hintergrund Dinge sabotiert hat, daß sie schiefgehen mußten – ohne jedoch selber in Verdacht zu kommen. Menschen wissen halt, wie die Dinge so gestaltet werden müssen, damit ein unvorhergesehenes Ereignis tatsächlich eintritt, welches als Rechtfertigung herangezogen werden kann. Bei der Citybank wurden daher drastische Maßnahmen ergriffen, damit so etwas keinesfalls mehr passiert: Wer seine Versprechen wiederholt nicht einhält - Abmahnung und dann - patsch – Kündigung! Aber Vorsicht! Bei „Mobbing“ werden oft die falschen beschuldigt, während die wahren Täter fast immer unbehelligt bleiben, weil sie scheinbar unbeteiligt sind, aber im Hintergrund tatsächlich Fäden ziehen. Hier passieren oft Soziodynamiken, die nur sehr schwer durchschaubar sind:

<http://www.little-idiot.de/teambuilding/AusweichlogikerInUnternehmen.pdf>

Intrinsische Motivation

Besser, als Kontrolle von oben, ist das Modell, welches Taiichi Ohno, der Kaizen – Guru bei Toyota eingeführt hat. Bei 100 Arbeitsprozessen für die Herstellung eines Auto's potenzieren sich stets die Fehler. Egal, wie niedrig die Fehlerquote je Arbeitsgang war, es wurde kein Auto fehlerfrei hergestellt. Daher führte er ein einfaches Prinzip ein – jeder nachfolgende Arbeitsgang hatte die Fehler des vorhergehenden zu kontrollieren und ggf. zu beseitigen. Resultat – alle Auto's liefen recht bald fehlerfrei vom Band. Zusammen mit Prämien für Fehlerfreiheit setzte die Führung auf „intrinsische Motivation“, die – anders als die „extrinsische Motivation“, Motivation von außen, von innen her, also aus eigenem Antrieb kommt. Die Mitarbeiter setzten sich in den Pausen zusammen, diskutierten Möglichkeiten der Fehlervermeidung – aber nun im Team – jeder hatte sich nicht mehr seinem Vorgesetzten gegenüber zu verantworten, sondern dem Team gegenüber, wo jeder jeden unter Druck gesetzt hat. Ein Fehler bedeutete Verlust der Prämie für das ganze Team. Dieses Prinzip funktioniert in sehr vielen Bereichen – so hat z.B. die Papierfabrik Zander in Düren schon Ende der 80er Jahre eine Prämie für alle Betriebsschlosser gezahlt, wenn diese defekte Maschinen besonders schnell, also schneller als die gemeinsam von GF und Mitarbeitern definierte „Regelreparaturzeit“ wieder in Gang gesetzt haben. So wurden am Jahresende jedem Betriebsschlosser zusätzlich zum Weihnachtsgeld noch einige hundert bis tausend Euro zusätzlich bezahlt. Es gibt zahlreiche Methoden, die Mitarbeiter „intrinsisch“ zu motivieren, leider wird das Wissen hierüber an keiner Universität vermittelt. Natürlich gibt es Fälle, wo dieses Prinzip nicht funktioniert hätte: Wenn zwischen zweien die „Chemie“ nicht stimmt, kein Gemeinsamkeitsgefühl aufkommen will. Hier hilft nur eines – Austausch. Vielleicht fällt hier die Parallele zu OpenSource Code auf – Jeder hat Überblick über den gesamten Quellcode, jeder darf an jeder Stelle Codestrukturen verändern, jeder fühlt sich in der Gemeinschaft mit verantwortlich für alles, denkt folglich viel intensiver für andere auch mit (nicht, daß bei Kollegen dauernd Programme auseinanderfliegen), man kommuniziert freiwillig viel intensiver, das Problem der Reaktanz, des Verschanzens hinter Zuständigkeitsbereichen, Abteilungsdenken, Pöschchencken, u.s.w. tritt nicht auf.

Nichts ist schlimmer, als wenn jemand zuständig für ein Modul und gleichzeitig Eigenbrötler ist, Mister „Unentbehrlich“ ist, und während seines Urlaubes, Krankheit nix mehr geht, oder bei seiner Kündigung sogar der komplette Code eingestampft werden muß, weil niemand sich mehr darin auskennt, dieser unleserlich, unstrukturiert ist,

ohne Dokumentation. Hier hilft nur – parallel das Modul neu entwickeln zu lassen, den Eigenbrötler, der partout nicht will – hinaus-schmeißen.

Meta – Prozesse

Agile Programming (AP) oder auch Agile Software Prozesse sind der neueste Schrei in der Softwareentwicklung. Agile Software Entwicklungsverfahren sind bewegliche, flinke Prozesse, die sich explizit nicht gegen Änderungen der ursprünglichen Anforderungen stellen. Am Projektanfang definierte Anforderungen stimmen zum Projektende erfahrungsgemäß nicht mehr. Hierdurch entsteht ein sog. „moving target“, also „variable Zielstellung“, die es gilt, so gut wie möglich zu treffen. Der Erfinder Coldewey teilt die Prozesse des AP daher in Meta-Prozesse (meta= (griech.) über) auf, die nur die Rahmenbedingungen stellen, und die konkrete Programmierarbeit auf.

AP verfolgt vorrangig folgende vier Maximen:

1. Einzelpersonen und Interaktionen sind wichtiger als Prozesse und Werkzeuge
2. Laufende Systeme sind wichtiger als umfangreiche Dokumentation
3. Zusammenarbeit mit dem Kunden ist wichtiger als Vertragsverhandlungen
4. Reagieren können auf Veränderung ist wichtiger als Pläne zu verfolgen

In Deutschland ist der Glaube an das Funktionieren von strengen Vorgaben für die Softwareentwicklung noch vorherrschend, begründet in der preußischen Tradition, welche das Harzburger Modell des hierarchischen, industriellen Führungsstils begründet: Befehl – Ausführung – Rückmeldung – Kontrolle. In der Softwareentwicklung ist das phasenorientierte „V-Modell XT“ vorherrschendes Modell, welches allerdings stark unter Druck geraten ist. Diese schwergewichtigen Software Entwicklungsprozesse, wie auch dem Wasserfall-Modell, dem alten V-Modell, oder dem Spiralmodell, konzentrieren sich sehr stark auf den einmaligen Durchlauf der Phasen Analyse, Design, Implementierung, Test und Auslieferung. Dies deutet darauf hin, dass die Software erstellende Industrie in ihrem Denken mehr den Projektgedanken als dem Produktgedanken behaftet ist. Auch der Gedanke, daß Software - Wartung nach der Fertigstellung erst beginnt, ist fehlerhaft, man könnte den Prozess der Softwarewartung bereits nach der Fertigstellung der minimalen Basisversion der Software ansetzen. Die Prozesse der Softwareentwicklung sind untrennbar mit dem Prozess der Wartung verbunden. Warum der Glaube an das V-Modell vorherrschend ist, obwohl in diesem die sich ständig verändernden Anforderungen an die Software nicht berücksichtigt sind, ist mir ein Rätsel. Der beste Prozess wiegt einen chaotischen Haufen von Anfängern, die kein professionelles Team bilden, nicht auf, wie die Open-Source Szene eindrucksvoll zeigt. Da professionelle Teams einen sehr hohen Grad an Selbstorganisation aufweisen, ist eine kleine Zahl an Methoden ausreichend. Zwölf auf einer Seite definierte Methoden bringen mehr als tausend Seiten Anforderungsprofile an die Software-Eigenschaften der beiden vorangegangenen Jahrzehnten, die eh nur von wenigen Experten überblickt und verstanden wurden. Man verabschiedet sich immer mehr von der Vorstellung, dass einheitliche wiederholbare Prozesse und detailliert planbare Projekte möglich sind. Auch die Vorstellung, durch klare Anweisungen und Kontrolle ein Projekt steuern zu können, schwindet, da die Erfahrung zeigt, daß kreative Prozesse sich nur schlecht befehlen und kontrollieren lassen. Wichtig ist es, ein Team mit kompetenten Mitgliedern zu haben! Man findet die Spezifikationen des V-Modells hier: http://www.kbst.bund.de/doc_-307632/V-Modell-XT-Dokumentation.htm

Beim Durchlesen kommt einem der Gedanke, daß hierbei Abteilungs- und Schubladendenken sowie der Projektgedanke vorherrschend ist, weit und breit kein „*prozessuales Denken*“, sprich „*emergente, kybernetische, rekursive, selbstkorrigierende Prozesse*“ in Sicht. Hierbei schaffen sich Bürokraten (Häuptlinge) erst ihre Existenzberechtigung, indem sie mit dem V-Modell XT äußerst aufwändige Vorarbeiten vorschreiben, die eh allesamt für die Katze sind, weil sich die Anforderungsprofile im Laufe der Softwareentwicklung erfahrungsgemäß noch gewaltig ändern. Oftmals nämlich wird während der Programmierung entdeckt, daß die prozessualen Abläufe im Unternehmen, der Organisation fehlerhaft, bzw. nicht wirklich verstanden sind. Nicht aufwändige Beschreibungen der Anforderungsprofile sorgen für gute Software, sondern nur wenige Handvoll einfacher Regeln führen zu sog. „emergenten Prozessen“, wie das Beispiel der Kontrollprozesse im Restaurant und das HTTPS-Protokoll zeigt.

Äußerst wichtig bei „Agiler Softwareentwicklung“ sind die sog. „*Metaprozesse*“, z.B. das bewußte und disziplinierte „Reflektieren“ über die Prozesse selber gehört dazu. Keine Regel ist unveränderbar, alles darf umgestaltet werden. Dieser Maxime folgt auch eXtreme Programming (XP).

Ein weiteres Beispiel für den kybernetischen Regeln folgende, emergente Softwareentwicklung ist **Adaptive Software Development (ASD)**. Jim Highsmith hat sich für seinen Prozess Anregungen aus dem Bereich der Biologie bei komplexen adaptiven Systemen geholt, genauer bei Umberto Maturana und seiner Erfindung „Autopoiese“, siehe „Baum der Erkenntnis“ Unter autopoietischen Systemen versteht man ein Zusammenspiel von Agenten, die mittels einfachen Regeln kommunizieren. Das dabei entstehende, emergente Verhalten zeigt folgende Eigenschaft: Einfache Regeln, gepaart mit komplexen Beziehungen, führen zu erfolgreichen Ergebnissen, während komplexe Regeln, gepaart mit wenigen Beziehungen, zum Scheitern führen. Zum weiteren Verständnigs der dynamischen Wechselwirkungen innerhalb von großen Netzwerken, siehe hierzu auch <http://www.little-idiot.de/teambuilding/KybernetikGesetzeDerNetze.pdf>. Bei der Entwicklung von Software betrachtet man nun Teammitglieder, Auftraggeber und Organisationen als Agenten. Die Regeln für die Zusammenarbeit bildet dabei das

Softwareentwicklungsverfahren. Somit entspricht ein Projekt einem komplexen, stets rückgekoppelten Regelsystem. Jim Highsmith hat folgendes Regelwerk erstellt:

1. Entwicklungsteam und Kunde starten gemeinsam mit dem Entwurf einer „*Vision*“
2. Team versucht innerhalb von vier Wochen möglichst viele Punkte der Prioritätenliste des Kunden zu erfüllen. Am Ende der vierwöchigen Iteration erhält das Team Feedback vom Kunden.
3. Während der Iteration wird dem Team vertraut - wenn der Plan nicht erfüllt werden konnte, geht man von einer Fehlplanung aus
4. Am Iterationsende überprüft das Team seine Leistung und verbessert den Prozeß.
5. Team plant nächste Iterationsrunde aufgrund vorangegangener Erfahrungen.
6. Management unterstützt das Team auf allen Ebenen.
7. Man überlässt die Auswahl der Arbeitstechniken dem Team.

Diese kleine Anzahl von Regeln führt zu äußerst flexiblen Projekten. Der ASD Lebenszyklus ist eine kurze Abfolge aus Spekulation, Kollaboration und Lernen.

Worin liegt hier die Emergenz? Diese Vorgehensweise sorgt dafür, daß z.B. vom Kunden im Laufe der Projektentwicklung gewünschten, besonderen „Features“, welche nicht elementar notwendig, und gleichzeitig komplex zu implementieren sind, also das Produkt unnötig verteuern würden, automatisch wegfallen. Kunde und Team können somit nach einfacheren, preiswerteren Alternativen suchen. Oftmals wird versucht, umständliche Prozesse der Verwaltung in der Software abzubilden. Analysiert man diese Prozessketten, bzw. deren mögliche Implementierung in die Software, so stellt sich oft heraus, daß hier in der Verwaltung durchaus noch einige Prozessketten gestrafft werden könnten, weswegen sich dann deren Implementierung in Software erübrigt. „Prozessuales Denken“ soll über Abteilungs- und Schubladendenken herrschen, nicht umgekehrt. 95% der im V-Modell XT enthaltenen Prüfspezifikationen, Berichte, u.s.w. könnten ersatzlos entfallen, das Projekt würde erheblich billiger werden, man kommt mit weniger Planung, Vorbereitung aus. Was dringend elementar benötigt wird, muß eh fertiggestellt werden, egal zu welchen Kosten. Es macht auch daher keinen Sinn, 30% der Gesamtkosten für überflüssige Planung bei sich ständig verändernden Anforderungen und Prozessmanagement aufzuwenden. Je nach Projektgröße schätzt man zwischen 15%-5% der Gesamtkosten für das Projektmanagement, leider wird nicht mitgerechnet, daß inzwischen auch die Programmierer mit erheblichem Verwaltungsblödsinn von ihrer Arbeit abgehalten werden.

Reaktanz

An dieser Stelle sei ein sehr wertvolles Experiment erwähnt „*Distrust – The hidden cost of control*“ von Armin Falk und Michael Kosfeld, siehe <http://www.little-idiot.de/teambuilding/TheHiddenCostOfControl.pdf> Die beiden Forscher wiesen anhand eines Modells nach, daß die von dem berühmten Soziologen und Kybernetiker Niklas Luhmann 1968 aufgestellte Annahme: „*Die selbsterfüllende Prophezeiung des Mißtrauens*“ sich bewahrheitete. Ihnen gelang es, bei einem Experiment mit Kontrolle von Mitarbeitern die Wirkung der Kontrolle zu isolieren. Ergebnis: Es zeigte sich, daß die Entscheidung, Mitarbeiter zu kontrollieren, den Willen, den Anweisungen der Geschäftsführung zu folgen, im Sinne der Firma zu handeln, dramatisch herabsetzte. Ursache: „**Reaktanz**“ - eine innere Grundhaltung, bei Druck zwanghaft immer das Gegenteil von dem tun zu müssen, was von einem verlangt wird, einer typischen Haltung von sog. „Ausweichlogikern“. Der Schuß ging gewaltig nach hinten los. **Dr. Joseph Murphys's Law** beschreibt die sich selbst erfüllende Prognose – Der Mitarbeiter, als eigenständig handelnder „Agent“ verhält sich dann oft nach dem Prinzip der sich selber erfüllenden Prognose so, wie das Management ihm ja durch die Einführung der Kontrolle unterstellt - Faul! Siehe <http://www.little-idiot.de/ihrseidallefaul.jpg>

SCRUM - Entwicklungs Prozess

SCRUM ist ein weiterer, dem „Agile Software Development“ verwandter, emergenter Prozess. Ken Schwaber und Jeff Sutherland holten sich für ihren Prozess Anregungen aus dem Bereich Verfahrenstechnik (Process Engineering). Die Idee ist, daß man zwischen wiederholbaren und empirischen Prozessen unterscheidet. Wiederholbare Prozesse sind solche, die vollständig verstanden sind, und deren Anwendung erfahrungsgemäß zu akzeptablen Ergebnissen führen, wohingegen empirische Prozesse chaotisch, sehr komplex, also nicht steuerbar sind. Das Hauptargument von Schwaber und Sutherland hierfür ist, daß Softwareentwicklung kein streng wiederholbarer Prozeß ist, da die Endprodukte immer verschieden sind. Empirische Prozesse müssen fortlaufend überwacht und angepasst werden, um das erwünschte, emergente Verhalten zu erzielen. Zur Steuerung ihres Prozesses verwenden sie den „Produktückstand“, womit die offenen Punkte der Prioritätenliste gemeint sind. Alle 30 Tage setzen sich Auftraggeber und Team zusammen und entscheiden, welche Punkte der Prioritätenliste in den folgenden 30 Tagen abgearbeitet werden soll. Das Team versucht nun, in diesen 30 Tagen, die als „Sprint“ bezeichnet werden, möglichst alle Aufgaben zu erfüllen. Aufgaben die nicht erfüllt werden konnten, bilden den „Sprint Rückstand“. Während des Sprints kann das Team völlig unabhängig und ungestört arbeiten. Gegen Ende des Sprints erfolgt ein „Sprint Review“, wobei die nicht erfüllten Aufgaben wieder auf den Produktückstand wandern. Nach einem Kaffeekränzchen, bei welchem Feedback und Erfahrungen ausgetauscht werden, beginnt wieder alles von vorne. Das Team hält täglich, immer zur gleichen Zeit, das sogenannte „Scrum

Meeting“ ab. Die groben Richtlinien für Scrum Meetings sind folgende:

1. Kurze Dauer von 15-30 Minuten
2. Was hat jeder seit dem letzten Meeting getan?
3. Was wird man bis zum nächsten Meeting tun?
4. Was hat bei der Arbeit behindert?
5. Lösungen werden nicht während eines Scrum Meeting gesucht oder diskutiert
6. Zu offenen Fragen werden kleine Teams gebildet, diese arbeiten nach Abschluss des Scrum Meetings weiter.
7. Scrum Meetings sind der Pulsschlag des Projektes

30 Tage hat der Auftraggeber die Möglichkeit, sich neue Anforderungen zu überlegen, die dann Teil des neuen Produktückstandes werden. Der Vorteil dieses Prozesses ist, dass das Team 30 Tage ungestört arbeiten kann. Die Durchführung der Aufgaben ist Teamsache, somit ist Scrum ein reiner Metaprozess. Scrum enthält keine Richtlinien für Programmierung.

Worin liegt hier die Emergenz? Scrum dient ausschließlich der Teambildung. Jeder erfährt, woran der andere arbeitet, was seine Probleme sind, und „fühlt“ in den Meetings die Stimmung seiner Kollegen. Man beginnt sich für die Arbeit des anderen zu interessieren, nimmt Anteil, sucht und gibt Ratschläge – Teambuilding. Scrum kann als psychologische Vorbereitung zu „Pair Programming“ verwendet werden, ein Aspekt des eXtreme Programming (XP).

Vollständige Entwicklungsprozesse am Beispiel XP

„eXtreme Programming“, entwickelt von **Kent Beck**, **Martin Fowler**, **Ward Cunningham** und **Ron Jeffries** ist voller emergenter Prozessen, welche differenzierte Methoden für die vier sich bis zur endgültigen Fertigstellung der Software ständig schnell wiederholenden, Entwicklungs-Zyklen (Iterationen) darstellen:

1. Planung, 2. Design, 3. Programmierung, 4. Test, 1. Planung, 2. Design

Dieses etwas merkwürdige, scheinbar ineffektive und aufwändige Prozedere begründet sich aus der Praxiserfahrung heraus: Ein Programmierer trägt täglich nur wenige, dauerhaft und korrekt geschriebene Codezeilen bei. **Nicht selten beträgt die Produktivität eines Programmierers über das ganze Projekt gemittelt nur ganz 5! Codezeilen je Arbeitstag.** Nichts ist beständiger, als die Veränderung - XP ist ein bewußter Prozess von Experimentieren und ständiger Verbesserung des Codes, wobei XP bewährte Methoden liefert, daß automatisch gute Software automatisch entsteht. Während in vielen Programmierer - Teams der innige Wunsch nach fehlerfreier, perfekter Software gepflegt wird, oder wie ein Damokles-Schwert über den Köpfen der Programmierer hängt, so berücksichtigt XP direkt von Anfang an, daß nichts und niemand perfekt ist, und Perfektion nur durch einen kontinuierlichen Verbesserungsprozess (KVP) entstehen kann. Diese neue Denke, bzw. das Bewußtsein in ein Team hinein zu tragen, ist ebenfalls ein Prozeß, der nicht von heute auf morgen passiert. Übrigends ist XP exakt die Implementierung / Umsetzung von KAIZEN, einem bei in Japan erfundenem „KVP“ (Kontinuierlicher Verbesserungs - Prozess), der zur Sicherung der Qualität im Fertigungsprozess und zur Optimierung der Fertigungs und Managementkosten (lean production, lean management, lean thinking, lean accounting) in der japanischen Autoindustrie, aber auch bei Boeing und bei Porsche angewendet wird. Mehr hierzu siehe <http://www.little-idiot.de/teambuilding/KaizenI.pdf>. Der Begriff TQM (Total Quality Management, Total=Allumfassend), in Japan geprägt, ist ein umfassendes Konzept, welches sich z.B. auch in der DIN/ISO Norm 8402 wiederfindet.

XP ist ein sehr raffiniertes Kaizen - Konzept, wobei jeder Punkt in den 4 verschiedenen Phasen, die zyklisch immer wieder durchlaufen werden, einen tieferen Hintersinn hat, also implizite Logiken enthält, die nicht direkt durchschaubar sind, weil die Effekte erst in der Dynamik sichtbar werden. Es ist für ein Teammitglied nur schwer zu verstehen, daß Prozesse, bei welchem jedes Teammitglied individuelle Nachteile hat, diese für das Team insgesamt jedoch von Vorteil sind. (emergente Effekte in kursiv...)

1. Planungsphase:

- User - Stories: Der Anwender beschreibt, basierend auf dem aktuellsten Zwischenrelease die Dinge, die er weiterhin benötigt, skizziert (anfangs laienhaft) ggf. weitere Arbeitsabläufe, ein verändertes Benutzer - Interface und sonstige Ideen frei auf Papier. *Diese dienen immer wieder der weiteren Abschätzung des Projekt - Umfangs, und der Festsetzung der weiteren Entwicklungs-Zyklen. Während der Implementierungsphase verändern sich stets die ursprünglichen Anforderungen, fehlerhafte Prozesse, oder Prozessketten, die gestrafft werden können, werden folglich auch nicht in Software übertragen.*
- Die Entwicklungs-Zyklen sind stets so kurz wie möglich zu halten. *Software - Releases werden mit ihren Eigenschaften immer wieder aufs Neue beschrieben und dies wird auch schriftlich festgelegt. Heraklit von Ephesos: „Nichts ist beständiger, als die Veränderung!“*
- Die Geschwindigkeit der Projektentwicklung wird in jedem Entwicklungs-Zyklus erneut abgeschätzt. *Hierbei fällt direkt auf, welche gewünschten, evtl. überflüssigen oder anders zu lösenden Features die*

Softwareentwicklung unnötig verteuern würden.

- Die Entwicklungszyklen werden alle 1-3 Wochen, aufgrund der sich ständig ändernden Anforderungen und Kundenwünsche, wieder und wieder erneut definiert. *Die Anforderungen verändern sich ständig, da der Kunde seine eigenen Prozesse im Unternehmen/Organisation während der Implementierung besser kennenlernt und parallel mit optimiert.*
- Die Planung der Entwicklungs-Zyklen findet immer wieder erneut statt. Hierbei werden ca. 15-20% der Gesamtzeit der Zyklen auf deren Planung verwendet. „Unit testing“ und „Refactoring“ werden ebenfalls mit eingeplant. *Die Qualität steht bei XP an alleroberster Stelle, je weniger Quellcode dabei entsteht, desto einfacher die Fehlerbeseitigung. Refactoring ist fester Bestandteil dieses Prozesses, Programmierer sind ständig damit beschäftigt, Code kürzer, effizienter und lesbarer zu gestalten.*
- Programmierer werden ständig getauscht (Rotationsprinzip). *Dies verhindert einen häufig auftretenden Effekt, daß aufgrund des Ausfalles einer Person im Team die gesamte Entwicklung zum Stillstand kommt. Dies trifft insbesondere auf Projekte mit hoher Komplexität und starker Koppelung der Module zu. Jeder kennt sich dann mit jedem Teil der Software aus und kann ggf. einspringen. Ggf. sollte Pair – Programming eingesetzt werden, womit dann keine Verzögerungen bei Ausfällen mehr auftreten. Programmierer können flexibel dann dort eingesetzt werden, wo es am meisten brennt. Höchste Priorität hat daher immer die Vermeidung von Wissens-Inseln im Team. Gerade an diesem Punkt gibt es die größten Widerstände, weil - jeder möchte sich unentbehrlich machen - aus Angst vor Jobverlust. Das Gegenteil jedoch ist der Fall: Wer die Denkweise von XP verinnerlicht hat, passt in jedes neue Programmierer-Team ... findet also immer einen Job. Auf flexible und erfahrene Programmierer mag und kann eh niemand verzichten.*
- Tägliche, kurze Meetings zum Arbeitsbeginn vermeiden längere Teamsitzungen bei Problemen. Die täglichen Aufgaben werden zugeordnet, Arbeitskapazitäten neu aufgeteilt. Hierbei kann es passieren, daß Programmierer die Arbeit ihres Kollegen fortsetzen müssen. Dies ist nur möglich, wenn sich jeder Programmierer im Gesamtquellcode genau auskennt, und natürlich auch Zugriff auf die Codestrukturen hat, diese sogar verändern/beräumen darf. Bis zu Teamgrößen von bis zu 20-30 Programmieren ist dies noch leicht machbar.
- XP - Regeln dürfen bei Auftreten von Problemen gebrochen werden. XP ist kein festgelegtes Prozedere, sondern darf - den Anforderungen und den Umständen entsprechend wohl begründet - abgeändert werden. *Die ständige Reflektion im Team darüber, ob entsprechend der Teamgröße oder Eigenschaften bestimmte Regeln noch sinnvoll sind, oder vorübergehend aufgehoben oder abgeändert werden, ist ebenfalls erwünscht. Heraklit von Ephesos: „Panta Rhei!“ - „Alles fließt!“*

2. Designphase:

- Einfachheit, KISS - Prinzip (Keep It Simple, Stupid!). Alles und jedes muß einfach durchschaubar sein, gut dokumentiert, dort, wo nötig. Der Hauptaspekt liegt stets auf der Austauschbarkeit des Programmierers. Jeder Programmierer muß sich sofort in der Arbeit eines anderen zurechtfinden können, und nach kurzer Einarbeitungszeit produktiv mitwirken können.
- Namen - das Schaffen von Bezeichnungen für Code - Abschnitte oder Programmen/Unterprogrammen erleichtert die Kommunikation im Team.
- CRC - Karten (**Class, Responsibilities, Collaboration**) dienen dem Design des Systems als Team. Hier sitzen alle Programmierer in einer Runde und halten Karten in der Hand (einer bedient die Mindmap Software am Beamer, siehe Data Beckers MindManager, 49 Euro!, **FreeMind** und **DoxyGen** reichen aber auch für größte Projekte völlig aus). Jede Karte repräsentiert ein Objekt mit Abhängigkeiten von anderen Klassen. Hierbei beginnt jemand in der Gruppe von Programmierern über seine Klassen und Abhängigkeiten zu reden, welches Objekt welche Nachrichten wohin sendet, u.s.w. Je mehr Personen bei diesem Prozess anwesend sind, umso besser. *Existieren zu viele Karten und Klassen, so wird die Zahl auf wenige je Person begrenzt. Warum? Ein Projekt darf maximal nur so komplex sein, wie jeder Beteiligte in der Lage ist, diese vollständig im Gedächtnis zu behalten. Hierbei werden in kurzer Zeit durch das gemeinschaftliche Denken Schwächen im Design entdeckt und korrigiert. Wird ein Design zu unübersichtlich oder zu komplex, steigt die innere Reibung im Team enorm an, die Zusammenarbeit wird sehr viel ineffizienter. Eine meiner beliebten Fragen zu Teambuilding bei Programmierern: „Erzähle mir genau, was Dein Kollege, der mit Dir am Modul programmiert, gerade tut!“ - Kann er hierzu keine Auskunft geben, so findet kein emergentes Miteinander füreinander mehr statt – ineffizient!*
- Bei technischen oder Design - Problemen programmiere ein einfaches Programm, welches das Problem unter Nichtberücksichtigung aller anderen Probleme löst. Da es nur Testzwecken dient, wird es später eh weggeworfen. *Das Ziel dieses Vorgehens ist es, das Risiko eines Fehldesigns zu reduzieren, und die Zuverlässigkeit der User - Story zu erhöhen. Wenn das Problem evtl. die Gesamtentwicklung verlangsamen könnte, so sollte Pair Programming eingesetzt werden, wobei zwei separate Entwickler sich eine oder zwei Wochen nur dieses Problems annehmen.*
- Füge niemals unnötig Funktionalität hinzu. Wir kennen alle das Problem, der Versuchung zu widerstehen, Dinge hinzuzufügen, weil sie das System verbessern würden. Wir müssen uns hierbei dauernd daran erinnern, bzw. selber disziplinieren, nichts zu programmieren, was nicht unbedingt benötigt wird. *Wenn nur ca. 10%*

*allen Codes bei XP tatsächlich überlebt, so verschwendet man 90% der Arbeitszeit. Man spart umso mehr Entwicklungszeit ein, je weniger Code für die Lösung der Aufgabe benötigt wird. Das höchste Prinzip, welches hier gilt, ist: „Number of Lines Not Written,“. Im Design jedoch berücksichtige stets die Möglichkeit, diese Funktionalitäten später hinzufügen zu *können*. Konzentriere Dich jedoch ausschließlich auf die morgendlichen Besprechungen und dein Tagespensum.*

- Der beste Programmierer - Team ist immer dasjenige, welches am wenigsten Code produzierte, eine hohe Funktionalität, Lesbarkeit und Wartbarkeit besitzt, und welches am Ende am wenigsten Code verwerfen mußte. Messbar ist so etwas nicht, gibt es eine Anreiz – Möglichkeit, Programmierer dazu zu bringen, so zu codieren? Ja, die gibt es: <http://www.little-idiot.de/teambuilding/PsychodynamischesProzessdesign.pdf>
- Refactoring, also der Prozess der dauernden Verbesserung der Code-Struktur muß immer höchste Priorität haben. Code muß einfach in seiner Struktur sein, leicht zu verstehen, zu modifizieren, und zu erweitern. Redundanzen sind aufzulösen, überflüssige Funktionen zu eliminieren, evtl. verworfene Designs können wieder verwendet werden. Jede Funktionalität darf nur einmal im Code vorkommen, doppelte oder ähnliche Funktionalitäten in ähnlichem Code werden zusammengefasst. *Der Abschied von seinem eigenen, bevorzugten Design zugunsten des aus Gründen der Praktikabilität durch Refactoring entstandenen, gemeinschaftlich entwickelten Designs fällt immer schwer. Manchmal muß man halt einsehen, daß das Ursprungsdesign ein guter Beginn war, aber nun obsolet ist.*

3. Programmieren, codieren:

- Der Kunde ist immer anwesend - elementarer Bestandteil des XP ist - er ist für die User - Stories verantwortlich, mit welcher die GUI ständig angepasst und verbessert wird - Er allein bestimmt das Aussehen und die Funktionalität der Software. Aufgrund der User Stories wird auch in jedem Entwicklungs-Zyklus immer wieder der Zeit- und Kostenrahmen erneut abgeschätzt. Dies ist insbesondere wichtig, wenn der Kunde neue Funktionalitäten einführen will, deren Notwendigkeit sich erst im Laufe des Projektes ergeben. Er bestimmt, wie das nächste, funktionierende Release aussehen soll. Er ist auch der einzige, der über Details Auskunft geben kann, die vergessen oder übersehen wurden. Desweiteren wird der Kunde immer bei Funktionalitätstests und Unit-Tests benötigt. *Interessant hierbei ist auch, daß direkt erkennbar ist, was Sonderwünsche kosten, und ob hierbei evtl. ein scheinbar „winziges“ Feature so aufwändig ist, daß der Etat gesprengt wird.*
- Es gibt für jede Programmiersprache im Internet sog. „Styleguides“, die festlegen, was wie bezeichnet wird, z.B. Pointer..., wie Code formatiert wird, sodaß die Lesbarkeit im Team erleichtert wird. Ziel ist es ja, daß jeder leicht den Code aller im Team lesen und verstehen kann, die Einarbeitungszeit gering wird. Refactoring muß erleichtert werden. *Unter dem Stichwort „best practice patterns“ finden sich sog. bewährte „Design Patterns“, die viel überflüssige Restrukturierung während der Entwicklungsphase vermeiden.*
- Unit Tests - Bevor auch nur eine einzige Zeile Code geschrieben wird, ist es unter XP Pflicht, zuerst eine Testroutine zu schreiben, die genau die Erwartung an ein Programm überprüft. Unit Tests für einfache Funktionen, Prozeduren oder Klassen zu schreiben, ist recht einfach. Schwieriger ist dies schon für Datenbankanwendungen, da Testdatensätze definiert werden müssen, noch schwieriger für GUIs mit Ein- und Ausgabe sowie Benutzerinteraktion. *Die Ursprungsidee einer Qualitätssicherung hat jedoch noch mehrere positive Nebeneffekte. Sie hilft dem Programmierer, sich bei dem Schreiben des Codes für die Unit sich nur auf das Wesentliche zu konzentrieren, sodaß nur so wenig Code geschrieben wird, wie benötigt wird, damit der Test erfüllt ist. Andererseits zeigt es anderen Teammitgliedern, wie eine Funktion, Prozedur oder Klasse verwendet wird, bzw. wofür diese da ist, ähnlich einem kleinen Programmierbeispiel oder Tutorial, wie man es aus dem Internet kennt.*
- Pair Programming - Jeder Code, der in ein Zwischenrelease oder endgültiges Release (production release) einfließt, wird von zwei Programmierern programmiert, die gemeinsam vor einem Computer sitzen und wechselweise programmieren. *Dies ist zu Beginn sehr gewöhnungsbedürftig, jedoch ist das Programmieren überhaupt ein schöpferischer Prozess, in welchem jeder stets seine Ruhepausen zum Nachdenken benötigt. Die Zeit des Nachdenkens darüber, wie nun etwas kodiert wird, sind oft viel länger, als die eigentliche Eingabe. So ähnlich, wie man während eines Gespräches Wortfindungsprobleme hat, hat jeder während des Programmierens auch mal ein „Brett vor dem Kopf,“. Derjenige, der dem anderen zuschaut, ist erst einmal aus der Verantwortung, kann dem Partner entspannt zuschauen, und sich dabei stressfrei auf den Code konzentrieren, der gerade geschrieben wird. Das Resultat ist, daß die Codequalität sehr viel besser wird, und im Endeffekt viel weniger Refactoring stattfinden muß. Es hat sich herausgestellt, daß Pair - Programming im Endeffekt die Kosten nicht erhöht.*
- Integration von Code - der Reihe nach, niemals parallel. Das unter sequential integration bekannte Einflechten von Code-Fragmenten in den entgültigen Code (production code) sollte nur wenigen Mitgliedern im Team der Programmierer vorbehalten sein. *Gerade dann, wenn von mehreren Entwicklern Code integriert werden soll, stellt man gewöhnlich fest, daß Code - Abhängigkeiten zu veralteten oder überflüssigen Codefragmenten existieren. Die Ursache liegt darin, daß alle Entwickler ja stets nie die neueste Codebasis kennen können, für welche sie Routinen entwickeln. Die hohe Abhängigkeit von Code untereinander, gerade in der frühen Phase*

der Entwicklung, macht die Sache sehr aufwändig. Je besser die Planung von Anfang an ist, je weniger Änderungen in der Struktur des Codes später stattfinden müssen, umso geringer ist die Wahrscheinlichkeit, daß Code verworfen werden muß, und mit ihm der davon abhängige Code. Schlechte Planung, Organisation und vor allem schlechtes Software - Design führen dazu, daß oft, obwohl die Zahl der Programmierer vergrößert wird, die Entwicklung noch mehr stagniert. Änderungen an Klassen, von denen viele andere Klassen abhängig sind, sind z.B. sehr aufwändig. Dies kann die Programmierleistung des Teams insgesamt dramatisch verschlechtern. Häufige Integration und schnelle Veröffentlichung im Team sind daher enorm wichtig. Definitionen von Schnittstellen zwischen Klassen und Reduktion der Codeabhängigkeit durch Schaffung von möglichst vielen, unabhängigen Modulen hat ebenfalls höchste Priorität. Das gesamte Konzept von J2EE / JBOSS basiert auf dieser Erkenntnis.

- Gemeinsame Eignerschaft am Code (collective code ownership) bedeutet, daß jeder Programmierer zu jeder Zeit Code anderer Team-Mitglieder korrigieren, verändern, erweitern oder bereinigen darf. Nur so wird verhindert, daß eine Einzelperson zu einer Art Nadelöhr für Veränderungen wird. Oft ist es so, daß eine kleine Verbesserung am Code eines anderen Team-Mitgliedes aufwändige Programmierung eines „Workarounds“ im eigenen Code einspart. Es gibt daher bei XP keinen Chef-Designer oder jemanden, welcher die „code changes“ mittels CVS integriert. Kein Mensch kann bei hochkomplexen Projekten alle Details im Kopf behalten. Außerdem, wenn das Team insgesamt für das Gesamtdesign des Codes verantwortlich ist, sollte jedes Mitglied auch das Recht haben, an jeder Stelle im Code Veränderungen oder Verbesserungen vorzunehmen. Höchste Instanz sind eh die Unit - Tests. Code, der diese Tests nicht erfolgreich durchläuft, darf eh nicht im production code eingeflochten werden. Umfangreiche Änderungen am Design erfordern automatisch Änderungen in den Test - Units, und man kann recht schnell entdecken, welche anderen Codefragmente plötzlich Test - Units nicht mehr bestehen. Da jeder im Team sich im Code anderer Team - Mitglieder auskennt, fällt auch das Ausscheiden eines Teammitgliedes kaum negativ auf. Daß jemand anderes in dem eigenen Code „herumpfuschen“ darf, ist zunächst jedem Programmierer höchst zuwider. Einerseits macht es auch Spaß, einem Kollegen „mal eben“ zu zeigen, wie es einfacher oder effektiver geht, und andererseits lernt man selber ja sehr viel dazu. Gerade junge Kollegen sollten diese Art von „Belehrungen“ als freundliche Geste auffassen, und sich für die Mühe ihres Kollegen bedanken - „Nobody is perfect!“. Es wird die Zeit kommen, wo man auch einem „alten Hasen“ mal neue Dinge zeigen kann ;-)

So kommt auch Spaß am Lernen und somit viel Dynamik in die Bude. Oft wird auf große Unterschiede bei der Fachkenntnis oder im Niveau der Programmierer im Team hingewiesen. Diese Wahrnehmung mag ja durchaus korrekt sein, jedoch ist hierbei zu bedenken, daß nicht jeder Mensch das große Glück hatte, gute Lehrer gehabt zu haben. Das, was wir sind, unsere Persönlichkeit, unsere Fähigkeiten haben wir nicht schon von Geburt an, sondern wir sind die Summe aller Einflüsse der Eltern, Familie und auch, ab dem Kindesalter beginnend, fremder Menschen in unserem Leben. Unsere Identität - auch Selbstbewußtsein genannt - ist die Summe aller Erfahrungen im Leben. Leider leben wir in einem Land, in welchem in preußischer Tradition Streitkultur und Demütigungskultur gepflegt wurde. In China hingegen herrscht z.B. eine „Konsenskultur“, mehr hierzu siehe auch <http://www.little-idiot.de/teambuilding/VonChinaLernen.pdf>
<http://www.little-idiot.de/teambuilding/DualistischAmbivalentDialektisch.pdf>

Durch die Anwendung von Pair-Programming und ständig neue Paarungen im Team lernen alle in sehr kurzer Zeit noch sehr viel hinzu, sodaß bald ein einheitliches Niveau im Team erreicht ist, wovon alle im Team profitieren, nicht nur fachlich, sondern insbesondere auch emotional. Unerfahrene Programmierer lernen dabei hauptsächlich Fachkenntnisse, die erfahrenen Programmierer lernen, wie man komplizierte Dinge mit einfachen Worten erklärt. Dies ist eine sehr hohe Kunst und stets eine Herausforderung auch für absolute Cracks. Die menschliche Biochemie ist so gestaltet, daß sowohl neue Erkenntnisse im Verstehen einer Tatsache, als auch das erfolgreiche Vermitteln mit Endorphin - Ausschüttung, also Glücksgefühlen „belohnt“ wird. Die Arbeit macht dann allen im Team sehr viel mehr Spaß, neue Potentiale werden entdeckt und freigesetzt. Hierbei kann jeder an sich selber beobachten, wie er an seinen Aufgaben im Team nicht nur fachlich, sondern auch von seinen persönlichen, menschlichen Qualitäten wächst. Dies gibt Freude im Leben, schafft Begeisterung, die ansteckend ist.

- Optimiere niemals während der Entwicklungsphase. Erst muß Code funktionsfähig sein, später dann können Nadelöhere beseitigt werden. Versuche niemals, zu erraten, wie groß die Verbesserung der Performance sein könnte, sondern messe sie. Aus der Kybernetik ist bekannt, daß komplexe, dynamische Systeme niemals lineares Verhalten zeigen. Siehe auch die 10 Netzgesetze: <http://beat.doebe.li/bibliothek/b00926.html> bzw. deren Erweiterung <http://www.little-idiot.de/teambuilding/KybernetikGesetzeDerNetze.pdf>
- Überstunden zerstören den Mannschaftsgeist und die Motivation im Team. Kann ein Termin nicht gehalten werden, so helfen auch keine Überstunden, oder die Vergrößerung des Teams. Die Ursache liegt darin, daß Programmierstätigkeit ein schöpferischer Akt ist, und ein Programmierer sehr viel mehr nachdenkt, als tatsächlich am Computer Code eintippt. Das Gehirn arbeitet hochgradig parallel, es denkt sogar im Schlaf weiter über Probleme nach. Nicht selten kennen wir Menschen die Lösung erst, nachdem wir eine Nacht drüber geschlafen haben. „Operative Hektik ersetzt geistige Windstille“ - dieser Spruch besagt, daß man durch Verbreitung von Unruhe im Team die Entwicklung insgesamt nicht beschleunigt, weil die Zahl der Fehler sich stark erhöht. Fehlersuche ist aber sehr viel aufwändiger und anstrengender, als wenn man Zeit

und Ruhe hat, vorher genauer nachzudenken, und dann fehlerfrei codiert. Jeder Fehler potenziert sich aufgrund der hohen Abhängigkeiten im Code, gerade in großen Teams. Stattdessen korrigiert man bei den „**release planning meetings**“ die Zielvorgaben, indem man sehr aufwändige Teile vereinfacht oder wegfällt läßt. XP ist so konzipiert, daß ständig Planung, Design, Codieren und Testen in schnellen Entwicklungszyklen stattfinden. Hierdurch werden frühzeitig unrealistische Ziele erkannt und korrigiert, bzw. es werden dann schnell Alternativen gefunden. Oft nämlich ist es einfacher, die Arbeitsabläufe im Unternehmen zu verändern, als umfangreich Software anzupassen.

4. Testphase:

- Jede Implementierung von Code wird durch Unit Tests geprüft. (Eigentlich ist Unit Testing eine Untermenge des „Test Driven Development = TDD“). XP ist extrem abhängig von Unit - Tests. Es gibt für verschiedenste Programmiersprachen ein sog. „unit test framework“, mit welchem man automatisierte Test Suites generieren kann. Code, der die Tests nicht erfolgreich durchlaufen hat, darf grundsätzlich nicht verwendet werden. Fehlt ein Test, so ist dieser unverzüglich zu erstellen. Der größte Widerstand gegen saubere Erstellung von Unit Tests ist stets gegen Ende der Entwicklung, kurz vor der Fertigstellung. *Hier zu schlampfen, wäre ein elementarer Fehler. Ohne vollständige Unit Tests dauert die Fehlersuche oft 100x so lange, wie sie eigentlich müßte, und außerdem muß die Software nach Fertigstellung des ersten Release ja auch weiter gewartet werden. UT zahlen sich immer aus, und zwar vielfach gegenüber dem Mehraufwand der Erstellung. Debugger finden nur einfache Programmierfehler, aber keine Logikfehler in den Abläufen bzw. dem Zusammenspiel der Klassen/Objekte. Unit Tests aber prüfen genau dieses Zusammenspiel. **Je härter es ist, einen Unit Test zu schreiben, umso mehr wird er auch tatsächlich benötigt, und umso größer wird die Zeitersparnis bei evtl. Fehlersuche sein!!!** Unit Tests sind noch wichtiger beim Refactoring. Nur so kann sichergestellt werden, daß Änderungen in der Code - Struktur zwecks Lesbarkeit und Wiederverwendbarkeit keine Änderungen in der Funktionalität bewirkt haben. Das frühzeitige Korrigieren von kleinen Fehlern in kurzen Intervallen spart viel mehr Zeit ein, als die Korrektur vieler Fehler kurz vor der Fertigstellung. Fehler potenzieren sich im Code, ebenso, wie der Aufwand ihrer Korrektur.*
- Wird ein Fehler entdeckt, so muß ein Test sicherstellen, daß dieser nicht noch einmal passiert. Der Unit Test muß dementsprechend angepasst werden.
- Akzeptanz - Tests (AT, früher Funktionalitäts - Tests genannt) sind das zweite Standbein von XP. Dies werden aus den **user stories** heraus geschaffen. Der Kunde beschreibt Szenarios, wie getestet werden soll, damit sichergestellt ist, daß die Funktionalität und Bedienbarkeit genau so ist, wie gewünscht. Jeder AT repräsentiert ein erwartetes Verhalten vom System. Der Kunde ist verantwortlich für Erstellung, Überprüfung und Einhaltung der Anforderungen an das System. Sobald mehrere Akzeptanz - Tests nicht erfolgreich beendet wurden, muß entschieden werden, welche Tests hohe, und welche niedrige Priorität haben. Eine user story gilt als nicht vollständig, wenn diese nicht alle Akzeptanz - Tests bestanden hat, was bedeutet, daß neue Akzeptanz - Tests geschrieben werden müssen, sobald bei einem Entwicklungs - Zyklus kein Fortschritt erzielt wurde. Qualitätssicherung (QA) ist ein wesentlicher Teil des XP Prozesses. Dieser kann durch eine unabhängige Gruppe durchgeführt werden, kann aber auch durch Mitglieder des Entwicklungsteams durchgeführt werden. Die zweite Lösung reduziert natürlich stark den Kommunikationsaufwand, weil Fehler nicht erst analysiert, dann niedergeschrieben und vermittelt werden müssen, sondern direkter Eingriff im Code das Problem dann egalisieren kann. Die Ergebnisse der Akzeptanz - Tests werden allen Team - Mitgliedern regelmäßig bekannt gemacht.

Nachdem dies erst einmal gesackt ist, so ging es mir, habe ich wieder von vorne angefangen, zu lesen, und so langsam dann erst verstanden, wie diese rekursiven, selbstkorrigierenden, dynamischen Prozesse eigentlich funktionieren, welche impliziten Logiken der Wechselwirkung der menschlichen Psyche mit den prozessualen Abläufen sich hinter jedem einzelnen Punkt verbergen.

Häufig wird die Frage nach „Kontrolle, Überprüfbarkeit“ der Leistung gestellt. Hierzu ist zu sagen, daß allein die Tatsache, in einem solchen „eXtreme Programming“ Team mitzuarbeiten, und die Dynamik täglich beobachten zu können, mit welcher Vehemenz, Intensität, Geschwindigkeit hier programmiert wird, sehr viel Freude macht, die ansteckt, also viel Endorphine freisetzt. Beim Menschen sorgt allein der Prozess der Erkenntnis für Endorphinausschüttung, siehe das Buch „Körper eigene Drogen“ von Joseph Zehentbauer. Nicht selten sieht das Großraumbüro, in welchem 1-2 Dutzend Programmierer wirken, wie ein Schlachtfeld aus. Papier, Diagramme allerorten, die Relikte intensiver Kommunikation. Durch Pair - Programming und ständigen Wechsel sind nach wenigen Monaten alle Teammitglieder wissensmäßig auf demselben Stand, die Unterschiede in der Leistung nur noch marginal. Software - Entwicklung im OpenSource - Bereich, also der Linux Kernel, die GNU Toolkits, allen voran der GCC C++-Compiler, die grafischen Benutzeroberflächen, der Apache Webserver, tausende von Anwendungswerkzeugen werden allesamt nach den Kriterien von XP entwickelt. Ein kleiner Unterschied jedoch besteht. Die Unit - Tests werden nicht von Programmen durchgeführt, sondern die gigantische Anwendergemeinde von vielen Millionen Usern, darunter viele hunderttausende Neugierige, die sich gerne eine Beta - Version von Linux (Mandrake Cooker, Debian Testing, ...) herunterladen, führen die Tests durch und melden, ob und wann ein Modul nicht funktioniert. Das „release early“

Prinzip der OpenSource Gemeinde sorgt dafür, daß frühzeitig Fehler bekannt und korrigiert werden. Das Content-Management-System ZopeX3, z.B., ist als erstes konsequent nach den Kriterien des XP entwickelt worden, inklusive dem Unit – Testing.

eXtreme Programming ist kein Meta-Modell, sondern ein ganz ausgefuchstes Prozedere, welches voller emergenter, also stets rekursiver, sich selber ständig verbessernder Prozesse ist, die letztendlich dafür sorgen, daß die typische, innere Reibung durch Kommunikation und Abstimmungsaufwand durch Fehlerfreiheit der Software von Anfang an, und der totalen Abwesenheit von möglichen Wissensinseln mit den typischen Folgekosten für Wartung, Qualitätssicherung überkompensiert wird. Fast alle Open-Source – Projekte werden (nicht vollständig) nach diesem Modell programmiert. „**eXtreme Programming**“ versucht auch, einen Paradigmenwechsel herbeizuführen. Je früher ein Fehler im Phasenverlauf von Analyse, Design, Implementierung, Test entdeckt wird, und je früher der Fehler im Phasenverlauf gemacht wurde, umso geringer sind die Fehlerbeseitigungskosten, und umso weniger Code muß verworfen werden. Dies ist letztendlich, über einen längeren Zeitraum betrachtet, Emergenz. Es wird viel weniger sinnlos an Konzepten weiterprogrammiert, die viel später eh verworfen werden müssen.

Vertreter der Phasenmodelle Wasserfall, V-Modell oder Spiralmodell versuchten deshalb, Entscheidungen möglichst früh zu treffen, Analyse und Design möglichst ausführlich und detailliert zu machen. Das Festhalten an früh getroffenen Entscheidungen führte bei diesen Modellen zu einem riesigen Berg an Dokumentation der einzelnen Phasen, die selten mit dem tatsächlichen Quellcode konsistent war. XP Vertreter versuchen die Kosten der Fehlerbeseitigung durch ein Bündel an Maßnahmen in den Griff zu bekommen. Dazu zählen die Praktiken einfaches Design, automatisierte Testfälle und Refactoring. XP Vertreter wehren sich nicht gegen Veränderung, sondern nehmen diese offen an. XP Vertreter sehen den Quellcode als zentrales Produkt und halten die Dokumentation in den verschiedenen Phasen so gering, wie möglich, da, wie die Praxiserfahrung zeigt, unvorhergesehen immer noch sehr viel verändert. Dokumentation wird in den Quellcode hineingeschrieben und mit ihm auch wieder weggeworfen. Papier indes bereinigt sich nicht selber. Der Aufwand der Nachführung der Paperware ist enorm - ständig inkonsistente Dokumentation, die abgeglichen werden muß. Stattdessen setzt XP auf Verbreitung des Wissens in möglichst allen Köpfen. XP Vertreter behaupten, dass durch das Maßnahmenbündel die Kosten für Änderungen und Fehlerbeseitigung sich in engen Grenzen hält, im Vergleich zu allen anderen Softwareentwicklungsmodellen. Was also zum Kuckuck macht unserer Bundesregierung da, die das V-Modell-XT „zwingend“ bei allen öffentlichen Projekten vorschreibt? Papierstapel generieren, und diese dann von links nach rechts schieben, unter ständiger Vervielfältigung? Typisch für die Denkweisen deutscher Beamter. Alles muß „kontrollierbar“ und „nachvollziehbar“ sein. Problem hierbei nur – ein Blick in den Quellcode würde auch genügen, könnte man diesen nur lesen

Viele der Softwareentwicklungsmodelle sind aus der Automobilindustrie übernommene, bewährte Methoden des „Kaizen“ Optimierungs – Gurus Taiichi Ohno, der das **Toyota Production System – TPS** erfunden, und damit das Unternehmen zu einem der lukrativsten Unternehmen überhaupt gemacht hat. Sein Motto: „*Mein Leben ist der ständige Versuch, gegen den gesunden Menschenverstand anzukämpfen*“ - Sprich – für einen einzelnen Mitarbeiter ist kaum einzusehen, daß Prozesse, die für ihn selber einen Mehraufwand bedeuten, für das Team insgesamt sehr viel mehr Effizienz bedeuten.

Nun ein Beispiel aus der Automobil - Produktion, welches hoffentlich die emergenten Prozesse erleuchtet, obwohl dies mit Softwareproduktion direkt nichts zu tun hat. Die Produktion eines hochkomplexen Automobils sehr vergleichbar mit der Produktion von Software, angefangen von Schnittstellen-Definitionen der hunderten logischen Einheiten, bis hin zu Baugruppen, die modular dezentral gefertigt, und auf einem Endmontageband nur noch „zusammengesteckt“ werden müssen. Schaut man sich die Zahlen von z.B. VW und Porsche an, so fällt auf, daß bei VW jeder Mitarbeiter in einem Jahr durchschnittlich 15 Auto's baut, Porsche jedoch nur 5 Auto's. Obwohl Porsche 3x soviel Mitarbeiter für ein Auto benötigt, erwirtschaftet Porsche einen 30x höheren Gewinn je Auto (23.000€/Auto Gewinn!). Folge – Porsche kauft langsam VW auf – 20% der Anteile nun im Besitz von Porsche. Das kleine Familienunternehmen Toyota hat letztes Jahr 10 Mrd. € Gewinn eingefahren, soviel, wie die Top3 Automobilhersteller zusammen. Renault bietet den „Logan“ handgefertigt aus den Produktionsstätten der tschechischen Firma „Dacia“, für einen Neupreis von 5000€ an, einem Preis, zu welchem VW mit Robotern noch nicht einmal einen einfachen Golf produzieren kann. Die impliziten Logiken völlig neuer Produktionstechniken der Automobilproduktion sorgen für eine Effizienz, welche beweist, daß die althergebrachten Denkweisen und Methoden völlig überholt sind.

Ursache für diese Gewinnexplosion bei Porsche und Toyota war das TPS (Toyota Production System), welches dafür sorgte, daß alle Auto's fehlerfrei vom Band liefen. Porsche gelang es erst 1994, also nach über 60 Jahren Automobilproduktion, den ersten fehlerfrei produzierten Porsche vom Band laufen zu lassen. Bei etwa 100 Arbeitsschritten, jeder zu 99% fehlerfrei ausgeführt, ergibt dies - bei einer Potenzierung der Fehler - über 30% Ausschuß am Ende des Bandes, was umfangreiche Nachbesserungsarbeiten zur Folge hatte. Taiichi Ohno hat hier eine Methode eingeführt, wo jeder nachfolgende Arbeitsschritt den vorhergehenden auf Fehlerfreiheit kontrolliert, bzw. diesen korrigiert. Genau diese implizite Logik der Qualitätskontrolle findet sich in den Methoden des XP (Extreme Programming) wieder, siehe oben, oder auch <http://www.little-idiot.de/xp/> .

Aus der Psychologie ist das Problem der Eigenwahrnehmung/Fremdwahrnehmung bekannt. Tollcollect hatte hier gleich mehrere Probleme. Erstens lief einem Projektleiter lief das Projekt aus dem Ruder. Er wußte nicht mehr, welcher

Programmierer woran genau arbeitete - diese kamen und gingen, wann sie wollten. Ein auf die Schnelle angeordnetes Kommunikationstraining nach dem Modell von „Friedemann Schulz von Thun“ schaffte auch keine Verbesserung, weil Mensch stets dazu neigt, nach erfolgreichem Training in die alten Verhaltensmustern zurückzufallen, insbesondere im Team. Die tatsächliche Ursache lag wo ganz anders – die Programmierer arbeiteten insgeheim für private Projekte – unbemerkt vom Management – Mangelhafte Kontrollprozesse waren die Kernursache.

Zweitens hatten mehrere Führungsmitarbeiter in dem „Zielvereinbarungssystem“ aufgrund von Druck von oben Versprechungen gemacht, die sie nicht einhalten konnten. Die Konsequenzen sind bekannt. Hauptproblem hierbei ist ein fehlerhaftes „psychodynamisches Prozessdesign“. Die Mitarbeiter entwerfen, programmieren, und melden Code als „fertig“ in einer Person. Die Kritische Distanz zu der eigenen Arbeit bzw. Leistung fehlt hier, Problem Eigenwahrnehmung/Fremdwahrnehmung. Daher ist es wichtig, die Softwareentwicklungsprozesse so zu designen, daß der Kollege quasi als „geistiges Korrektiv“ wirkt, dieser kritisch das „fertige Modul“ seines Kollegen testet, und dann unabhängig als „fertig“, „teilfertig“, „unfertig“ meldet. Auch sorgen zusätzliche Entlohnungssysteme dafür, daß ein Programmierer, der seine Zeitzusage nicht einhält, keine Prämie erhält, und eine höhere, wenn er vorzeitig fertig ist, und noch höhere Prämien gibt es, wenn die Gruppe vorzeitig fertig wird. Durch dieses einfache Prämiensystem, welches als „Psychodynamisches Prozessdesign“ eine ganze Kette von Verhaltensänderungen im Team bewirkt, habe ich in der Praxis Einsparungen von bis zu 40% der Entwicklungskosten mühelos erreicht, bzw. „bewirkt“. Dieses Prinzip ist ein kybernetisches Prinzip, siehe auch Schwarmverhalten oder Schwarmintelligenz unter: <http://www.little-idiot.de/teambuilding/KybernetikGesetzeDerNetze.pdf>

Die Methode des Aufspürens von Fehlern ist in der Software – Produktion als Unit-Test bzw. TDD (Test Driven Development) bekannt. Bevor nicht alle Module erfolgreich die Tests durchlaufen haben, darf nicht weiter programmiert werden. Die Parallele zur Qualitätskontrolle in der Automobilindustrie ist offensichtlich.

Unit – Tests erscheinen auf den ersten Blick recht aufwändig, jedoch werden sie umso dringender benötigt, je komplexer das Programm-Modul ist. Weil durch „moderne“ OO – Programmierung (Objekt - Orientierte Programmierung) mit Vererbung, Delegation, Polymorphie, Aspektorientierte Programmierung (AP) eine hohe Abhängigkeit des Codes untereinander (Code – Hierarchien) geschaffen wurde, sind elementare Änderungen an den Basisklassen fatal. Kleine Änderungen dort ziehen umfangreiche Änderungen an den abhängigen Klassen nach sich. Vergleicht man dies mit der Konstruktion eines Autos, so fällt z.B. bei einem VW-Bulli auf, daß nur zum Tausch einer einfachen Kupplungsscheibe der komplette Motor samt Vorderachse ausgebaut werden muß (VR6). Dieses Auto, z.B., ist aufgrund hoher Abhängigkeiten der Baueinheiten untereinander extrem wartungsfeindlich. Übertragen auf Software – Programmierung bedeutet dies, daß der Ehrgeiz der Programmierer, saubere Klassenhierarchien haben zu wollen, zwar für sauber aufgeräumten Quellcode sorgt, jedoch Änderungen der Codestrukturen grundsätzlich sehr aufwändig sind. Als Beispiel möge hier die grafische Benutzeroberfläche KDE von Linux dienen, welche bislang auf QT3 aufgebaut ist. Die Entwickler haben es nicht geschafft, QT3 „sanft“ zu QT4 zu migrieren, stattdessen haben sie QT4 von Grund auf neu programmiert und strukturiert. Kosten: Ca. 1/3 derjenigen von QT3, bei JBOSS, dem Clone von J2EE, lagen die Kosten ähnlich.

Allzusehr durchorganisierter Code kann gegenüber teilchaotischem Code erheblich wartungsaufwändiger sein.

Refactoring ist insbesondere dann, wenn die Codestrukturen zu Beginn des Programmierprojektes zu hierarchisch, sprich zu wenig „chaotisch“ sind, oft sehr teuer. Genau dieser Punkt ist ein häufiger Streitpunkt bei Software – Architekten – viele vertreten den Standpunkt, daß von Anfang an saubere Design – Pattern für ein perfektes Design sorgen müßten. Wohin das geführt hat, kann man z.B. beim Projekt „FISCUS“ der Bundesregierung sehen, wo 130 hochkarätige Programmierer mit modernsten UML – Werkzeugen (Rational ROSE) und TOP – Software – Architekten eine von Grund auf gute Software – Struktur geschaffen haben, mit „sauberen“ Klassen-Hierarchien. Leider nur ist bei einem solchen Mammut – Projekt kein bewährtes „Design - Pattern“ verfügbar gewesen. Modelle für ausgereifte Design-Pattern siehe: <http://www.cs.wustl.edu/~schmidt/> .

Um es noch einmal mit aller Deutlichkeit zu sagen. Der Wunsch nach einem „Design-Pattern“ nach welche die Software entwickelt wird, und welches daher zu Beginn mit Hilfe erfahrener Berater genau evaluiert und dann festgelegt wird, ist Schwachsinn!!!! Design-Pattern verändern sich im Laufe des Fortschrittes der Softwareproduktion. Das zu Beginn gewählte Designpattern muß durch flexible Strukturen schnelle Veränderungen erlauben, darf also nicht zu starr sein. Dann, wenn sich aufgrund der Anforderungen Strukturen herausbilden, stampft man dieses Pattern ein, und programmiert im Grunde alles noch einmal neu. Dabei jedoch haben alle Programmierer des XP – Teams das neue Bild der erheblich leistungsfähigeren und langfristig viel weniger aufwändigen und wartungsärmeren Struktur genau im Kopf. Das Neuschreiben einer Software flutscht dann wunderbar, kalkulieren kann man etwa mit 1/3 der bis dahin aufgelaufenen Kosten. Danach jedoch sinken die Kosten für die weitere Fertigstellung des Projektes durch die neuen Strukturen erheblich; es ist ein richtiger Schub im Fortschreiten des Projektes zu erwarten. Und selbst dann, wenn man nochmals den gesamten Code komplett einstampfen muß, und selbst dann, wenn die Kosten die Planung dann überschreiten, so erhält man diesmal dann eine Codestruktur, die langfristig sehr viel Kosten für Qualitätssicherung und Wartung (Refactoring) einspart, einfach erweiterbar und pflegeleicht ist. Darüber hinaus stellt sie dann mit hoher Wahrscheinlichkeit die optimale Lösung für den Auftraggeber dar.

Ich verstehe sehr den Wunsch, ein solches Design-Pattern von Anfang an zu haben, dieser hat sich jedoch als nicht sehr wirklichkeitsgerecht geschweige denn faktizitätstreu herausgestellt. Was ich hier beschreibe sind Erfahrungswerte aus

der Wirklichkeit, keine Träumereien von Anhängern des V-Modell-XT oder ähnlichen Papiertigern. Der von Kent Beck in „eXtreme Programming“ vorgeschlagene Ansatz, Pattern langsam zu entwickeln, wobei von Zeit zu Zeit bewußt der gesamte Quellcode eingestampft, und von Grund auf neu programmiert wird, ist immer erfolgreicher. Nach mehrmaligem Neuprogrammieren des Codes zu Beginn des Projektes hätte das Projekt FISCUS sicher schnell eine Codestruktur erhalten, welche dauerhaft brauchbar und wartbar gewesen wäre. Der Apache Webserver, die Portal – Software PHP-Nuke, das CMS – System Zope, die Groupware „E-Groupware“, PHPProject und viele andere Open-Source Software – Projekte zeigen klar, daß sogar sehr viel einfachere Software mehrfach neu programmiert wurde, weil irgendwann Erweiterungen und Änderungen unmöglich wurden. Als einziges Projekt ist Zope vollständig nach den Methoden des XP programmiert wurden, inclusive umfangreichen UNIT-TESTS. Nicht zuletzt sind z.B. auch QT3, KDE, Zope, Linux, Apache im Laufe der Jahre komplett eingestampft und völlig neu entwickelt worden. Linux 1.x und 2.x haben nicht eine einzige Codezeile mehr miteinander gemeinsam. QT3 mußte, obwohl in C++ perfekt sauber strukturiert gewesen, komplett „from scratch“ neu programmiert werden, bei bedingter Kompatibilität zum Vorgänger.

Meta – Modelle am Beispiel des CMMI - Modells

CMMI ist die Beschreibung eines **Meta-Prozesses** zur Softwareentwicklung, welcher gute, fehlerfreie, lesbarer und wartungsarme Software hervorbringen soll. Im Gegensatz zu einer konkreten Prozessbeschreibung, wie z.B. XP **definiert CMMI nur die Anforderungen an eine gute Produktentwicklung** (das „was“), gibt aber keine konkreten Schritte vor (das „wie“).

Das primäre Ziel von CMMI als Meta-Prozess ist es, für einen kontinuierlichen Wandel der eigentlichen Software – Herstellungsprozesse zu sorgen, ist also genaugenommen nur ein Prozess zur Unterstützung des KVP, also dem „kontinuierlichen Verbesserungs - Prozess“, indem flexibel stets neue Anforderungen bzw. Kriterien an eine professionelle Produkt-Entwicklungs-Organisation definiert werden. „Panta rhei“ - sagter Heraklit - „Alles fließt!“.

Die Definition der Art und Weise des Entwicklungsprozesses obliegt der Organisation selber, und ist eine wichtige Teilaufgabe der Prozessverbesserung. Da CMMI keinen konkreten Entwicklungsprozess definiert, kann CMMI auf sehr unterschiedliche Organisationen und Organisationsgrößen angewendet werden.

Eine besondere Eigenschaft von CMMI ist, dass dieses nicht nur die Entwicklungsprojekte an sich adressiert, sondern auch die projektbezogenen Aufgaben der Organisation (z.B. Bereitstellung von Ressourcen, Durchführung von Trainingsmaßnahmen). Ein weiteres besonderes Merkmal ist, dass CMMI sehr viel Wert auf den gelebten Prozess legt, im Gegensatz zum häufig als „**Schrankware**“ bezeichneten, **dokumentierten, aber nicht gelebten Prozess**.

Aufbau des Modells

CMMI definiert eine Reihe von Prozessgebieten (z.B. Projektplanung, Anforderungsentwicklung, also die organisationsweite Prozessdefinition). Ein Prozessgebiet spezifiziert die Anforderungen an eine professionelle Produkt-Entwicklung in einem bestimmten Gebiet durch ein Bündel verwandter Praktiken, die, sofern sie gemeinsam ausgeführt werden, eine Reihe von Zielen erfüllen, die für eine deutliche Verbesserung auf diesem Gebiet wichtig sind.

Beispiel: Beim Prozessgebiet „Projektplanung“ sind die Ziele „Schätzungen aufstellen“, „Einen Projektplan entwickeln“ und „Gegenseitige Verpflichtung auf den Plan herbeiführen“.

Die Praktiken zum Ziel „Schätzungen aufstellen“ sind „Umfang des Projekts schätzen“, „Attribute der Arbeitsergebnisse und Aufgaben schätzen“, „Projektlebenszyklus definieren“ und „Schätzungen von Aufwand und Kosten aufstellen“. Für die Prozessgebiete, Ziele und Praktiken gibt CMMI jeweils zusätzliche erklärende Informationen. So wird z.B. jedes Prozessgebiet zunächst erläutert, und dann werden damit in Verbindung stehende Prozessgebiete aufgezählt. Jede Praktik wird durch Erklärungstext, durch typische Arbeitsergebnisse und durch typische Arbeitsschritte weiter erläutert. Diese Hinweise sollen bei der Umsetzung helfen.

Die Prozessgebiete sind in vier Kategorien eingeteilt: **Projektmanagement, Entwicklung, Unterstützung** und **Prozessmanagement**. Während die ersten beiden Kategorien die Prozessgebiete enthalten, die typischerweise in Projekten umgesetzt werden, ist Prozessmanagement vor allem eine organisationsweite Aufgabe. Die Prozessgebiete in der Kategorie Unterstützung können sowohl eine Projektaufgabe als auch eine Organisationsaufgabe sein.

Neben den Praktiken, die spezifisch für ein Prozessgebiet sind, adressiert CMMI auch explizit die Institutionalisierung der Prozesse. Mit „Institutionalisierung“ ist gemeint, dass die Prozesse in der Organisation selbstverständlich und als Teil der täglichen Arbeit gelebt werden. Insbesondere in Zeiten von Stress haben institutionalisierte Prozesse Bestand. Neben den Praktiken, die spezifisch für die einzelnen Prozessgebiete sind, definiert CMMI Praktiken, welche die Institutionalisierung umsetzen. Diese Praktiken zur Institutionalisierung werden als generische Praktiken bezeichnet (da sie für alle Prozessgebiete gleich sind). Die Umsetzung vieler generischer Praktiken ist eine Aufgabe der Organisation.

Fähigkeitsgrade und Reifegrade

CMMI adressiert die Verbesserung innerhalb eines Prozessgebiets durch sogenannte Fähigkeitsgrade (capability levels). **Ein Fähigkeitsgrad bezeichnet den Grad der Institutionalisierung eines einzelnen Prozessgebiets**. Die Fähigkeitsgrade sind:

- 0 - Incomplete - Ausgangszustand, keine Anforderungen**
- 1 - Performed - die spezifischen Ziele des Prozessgebiets werden erreicht**
- 2 - Managed - der Prozess wird gemanagt**
- 3 - Defined - der Prozess wird auf Basis eines angepassten Standard-Prozesses gemanagt und verbessert**
- 4 - Quantitatively Managed - der Prozess steht unter statistischer Prozesskontrolle**
- 5 - Optimizing - der Prozess wird mit den Daten aus der statistischen Prozesskontrolle verbessert**

Neben den Fähigkeitsgraden eines einzelnen Prozessgebiets definiert CMMI Reifegrade (maturity levels). *Ein Reifegrad umfasst eine Menge von Prozessgebieten, die zu einem bestimmten Fähigkeitsgrad umgesetzt sein müssen.*

Diese Reifegrade sind:

- 1 -Initial - keine Anforderungen, diesen Reifegrad hat jede Organisation automatisch**
- 2 -Managed - die Projekte werden gemanagt durchgeführt und ein ähnliches Projekt kann erfolgreich wiederholt werden**
- 3 -Defined - die Projekte werden nach einem angepassten Standard-Prozess durchgeführt, und es gibt eine kontinuierliche Prozessverbesserung**
- 4 -Quantitatively Managed - es wird eine statistische Prozesskontrolle durchgeführt**
- 5 -Optimizing - die Prozesse werden mit den Daten aus der statistischen Prozesskontrolle verbessert**

Die Bewertung des Reifegrades bzw. der Fähigkeitsgrade einer Organisation geschieht durch autorisierte, erfahrene Personen.

Abgrenzung CMMI zu anderen Normen

Im Unterschied zur DIN EN ISO 9001 ist CMMI nur für den Produkt-Entwicklungsprozess entwickelt worden, während die DIN EN ISO 9001 die gesamte Organisation, und damit mehr die Breite abdeckt. CMMI setzt die Anforderungen der gerade im März veröffentlichten Norm ISO 15504 mit ihrem in Teil 5 enthaltenen Bewertungsverfahren **SPICE** an ein Prozessmodell um. Neben CMMI gibt es z.B. die Norm ISO 12207 und Norm ISO 15228, die ebenfalls ein Prozessmodell für Softwareentwicklung sind. Im Gegensatz zu CMMI gehen diese beiden Normen aber nicht über die Definition der Titel der Praktiken von CMMI hinaus, es fehlen die umfangreichen Erklärungen aus CMMI, und es gibt auch keine Integration der beiden Normen. Inhaltlich fordern ISO 12207 und ISO 15288 im wesentlichen das Gleiche wie CMMI. Literatur siehe [24], [25], [26].

Analyse und Beurteilung von Softwarestrukturen

Typischerweise befindet man sich in einem laufenden Projekt der Softwareentwicklung und möchte nun auf emergente Software-Entwicklungsprozesse umstellen, und es folgt fast immer der Wunsch nach einer Bestandsaufnahme – die *dämliche* Frage nach dem „Status Quo“ - Wo stehen wir, wohin wollen wir?

Wer diese Frage stellt, hat offensichtlich noch Probleme mit dem Umdenken. Wir wollen weg vom „Statusdenken“, „Imagedenken“, „Abteilungsdenken“, „Zuständigkeitsdenken“ - hin zum **„Prozessualen Denken“**, dem Denken ausschließlich in emergenten Prozessen, welche ja schleichend dafür sorgen, daß bessere Software entsteht, und zwar automatisch im Laufe der Zeit nach der Umstellung auf neue Entwicklungsprozesse. Also noch einmal zurück zum Anfang dieses Textes!

Auswirkungen von Codestrukturen auf Unternehmensschicksale

Dennoch kann man aus den Fehlern der Vergangenheit lernen – wo kann man mehr aus den Fehlern bei der Entwicklung riesiger Softwarepakete lernen? Bei OpenSource – Software – Entwicklungsprozessen! Nur so als Spielerei: Wie analysiert und beurteilt man vorhandene Code – Strukturen?

Jesús M. González-Barahona et al. haben ein Programm geschaffen, welches aus einem CVS – Baum - siehe auch „DOXYGEN“ - heraus mit Hilfe des genetischen **Girvan - Newman** - Algorithmus die Codestrukturen visualisieren kann.

Anhand des COCOMO Modells wurden z.B. die geschätzten Kosten einer Neuentwicklung der Debian Linux 2.2 Distribution auf 1.9 Billionen US\$ geschätzt. IBM hat schon lange verstanden, und schlicht jede weitere Entwicklung eigener Betriebssysteme eingestellt, auch Microsoft kümmert sich neuerdings (November 2006) um LINUX durch Partnerschaftsverträge mit NOVELL, empfiehlt sogar Linux bei seinen Kunden. Was steckt dahinter? Microsoft plant, aus dem Geschäft mit Server – Betriebssystemen mittelfristig auszusteigen. Laut Auskunft eines ehemaligen, hochrangigen Software-Architekten schaut es mit dem neuen Windows Vista Betriebssystem und den „neuen Innovationen“, so z.B. der Idee, den SQL Datenbankserver als Filesystem zu verwenden (AS/400 hatte immer schon

DB2 als Filesystem, Linux kann mit MySQL als „Filesystem“ betrieben werden), sehr mies aus. Das Konzept ist laut seiner Aussage nicht performant, funktioniert so nicht, das System wäre zu komplex, zu sehr fehlerbehaftet. Nunja – wahrscheinlich ist seine Aussage schon, meiner Meinung nach, lassen wir uns überraschen ;-)

Wer sich den Wandel der Apache – Codestrukturen, mit Hilfe des GN-Algorithmus visualisiert, einmal anschauen möchte: <http://www.little-idiot.de/teambuilding/apache-structure.pdf>. Ich habe eine Unzahl von „Snapshots“ von OpenSource – Software gemacht, am beeindruckendsten sind die Entwicklungsfortschritte des Apache – Webservers, dem absoluten Marktführer weltweit.

Lernen aus OpenSource Entwicklungsfehlern

Von den Entwicklern der QT3 und QT4 Libraries (welche hinter der KDE – Benutzeroberfläche von Linux stecken) wissen wir, daß die komplette Neuentwicklung von QT4 unter Berücksichtigung des Wissens um die Strukturfehler der neuen Version QT4 nur etwa 1/3 der Kosten von QT3 gekostet hat. Mit Stolz verweisen die Entwickler darauf, daß QT4 nun erheblich wartungsfreundlicher ist, bzw. auch viel einfacher zu erweitern bzw. zu ergänzen ist. Software-Entwicklern, die Software Cross - Platform - Entwickeln wollen, sei QT4 empfohlen – sie steht für Windows, MAC OS X, Linux, UNIX zur Verfügung.

Man muß hierbei allerdings berücksichtigen, daß die Entwicklung einer GUI sehr saubere Klassen - Hierarchien erfordert, da die Einarbeitungszeit für Entwickler recht hoch ist. Die Entwicklung einer GUI ist nicht vergleichbar mit dem Projekt FISCUS. Ich vermute mal, daß seit Entdeckung der Tatsache, daß etwas mit dem Design des FISCUS - Projekt nicht stimmte, und dessen Einstellung - viele hundert Mannjahre unsinnig verballert wurden. Niemand der Entscheider hatte den Mut, den Code vor Erreichung der 5 Jahre sang – und klanglos einzustampfen, und von vorne zu beginnen: „*Es kann nicht sein, was nicht sein darf!*“ - Niemand hat den Mut gehabt, zuzugeben, daß das Team – einschließlich ja einem selber – versagt hat.

Was nutzt es, nach 3 Monaten im Team von 50 Leuten festzustellen, daß der eingeschlagene Weg nicht gangbar war, oder die Komplikationen absehbar immer größer werden, und dann noch weiterzumachen? Der Code muß eh eingestampft werden, wozu also noch mehr Geld verbrennen? Das Schicksal z.B. ereilte den Linux Kernel. Der Linux Kernel 1.x wurde komplett eingestampft, weil die Codeabhängigkeiten zu groß wurden, insbesondere bei den Treibern. Daher wurde zuerst eine Treiberschnittstelle geschaffen, die so gewaltige Änderungen am Kernel nach sich zog, daß der Kernel 2.x nichts mehr mit Kernel 1.x zu tun hatte, außer dem Namen. Auch hat Linus Torwalds den TCP/IP Code komplett entfernt und durch BSD – Code ersetzt. Vergleicht man z.B. mit den Methoden des XP, so fällt auf, daß XP alle vier Entwicklungszyklen „1. Planung“ „2. Design“, „3. Codieren“, „4. Testen“ in schneller Folge von ca. 1-2 Wochen immer wieder erneut durchlaufen werden, iterativ. Der eigentliche Zweck dieser Vorgehensweise war ja, siehe oben, daß Designfehler konsequent frühzeitig entdeckt und radikal korrigiert werden, egal, ob der Code zu diesem Zeitpunkt komplett neu geschrieben werden muß, oder nicht.

Bei OpenSource Projekten, wie z.B. PHPNuke kann man in Google nachlesen, daß recht frühzeitig schon eine Abspaltung des Codes in Postnuke stattgefunden hat, aufgrund der mangenden Modularität und Sprachanpassungsmöglichkeiten. Recht viele Projekte in der Open-Source – Szene wurden einfach eingestampft, und neu begonnen, wie Eric Raymond beschrieben hat: <http://www.little-idiot.de/xp/opensourceentwicklung.htm> (siehe auch: „*Die Kathedrale und der Basaar*“)

Dafür besitzen die erfolgreichen Open-Source Programme ein glänzendes Design (-Pattern), sind modular aufgebaut, einfach zu erweitern und zu verändern.

Wann Refactoring scheitern muß – mit fatalen Konsequenzen

Die Grenzen des „Refactoring“ sind recht schnell erreicht, wenn die Codestrukturen bzw. das Design nicht stimmen, z.B. zu hohe Abhängigkeiten untereinander. Diese müssen immer zuerst komplett aufgelöst werden, damit es überhaupt weitergehen kann. Nirgendwo kann man mehr über Komplexitätstheorie in der Softwareentwicklung lernen, als in der Open-Source – Szene und dort vor allem bei den besonders großen Softwarepaketen, wie z.B. dem Linux – Kernel, dem Free/NetBSD Kernel, dem Apache – Webserver mitsamt seinen unzähligen Modulen, dem Office – Paket OpenOffice 2.0, Netscape/Mozilla und einigen Groupware – Produkten, darunter z.B. auch Compière, einem ERP / CRM – Programm.

So haben z.B. die Besitzer der kommerziellen „Closed Source“ - Software CYCOS, siehe <http://www.cycos.de>, nach sorgfältiger Analyse der Code - und Programmierer – Strukturen, verteilt über mehrere Länder, darunter auch Frankreich und U.S.A. - schnell und schmerzlos entschlossen, ihren Laden an Siemens zu verkaufen. Ende, Schluß, „aus die Maus“. Refactoring, so ergab die Analyse, war aufwändiger, als eine komplette Neuprogrammierung. Neuprogrammierung war leider auch nicht mehr möglich, weil das Wissen um die Kommunikations - Protokolle durch den Wechsel und Ausscheiden von Programmierern in der Firma nicht mehr vorhanden war. Niemand weiß nun mehr genau, wie die Software insgesamt funktioniert, wie und wo welche Kommunikationsprotokolle was bewirken, wie sie implementiert sind. Das Wissen ist unwiederbringlich verloren, fehlerhaftes Wissens-Management.

Auch ein aus den U.S.A. eingeflogener Guru für SCRUM (Unterart des „Agile Programming“) konnte an dieser Tatsache nichts ändern. Ein ähnliches Schicksal hatte auch die U.S.A., genauer die NASA getroffen – es gab

niemanden, der das Ingenieurwissen der Gruppe um Werner von Braun herum festgehalten hatte. Aus der Raumfahrer-Nation, welche einst zum Mond flog, ist ein teurer Papiertiger namens NASA geworden, ohne eine wirklich sinnvolle Aufgabe. Russische und europäische Raketen bringen Satelliten viel preiswerter und sicherer ins All, eine Raumstation ISS benötigt niemand, die Industrie hat auch kein Interesse, weil immer alles auf der Erde produziert werden muß.

Ein ähnliches Schicksal ereilte die IBM – Truppe, die bei INA Nadellager eine Lagerverwaltung programmieren, und an SAP/R3 anschließen sollte. Der 2-stellige Millionenbetrag wurde vollständig verbrannt, die Projektleiter gaben sich die Türklinke in die Hand, zuletzt wurden immer mehr Programmierer ins Team „geschmissen“, das Projekt scheiterte. Niemand konnte mehr die einfachsten Fragen beantworten: **„Wenn ich hier im Modul etwas an dem Interface ändere, welche Auswirkungen hat das, an welchen Stellen muß Code in welchem Umfang geändert werden?“**.

Insgesamt kann man sehr viel von der Entwicklung erfolgreicher Open-Source – Projekte lernen. Extreme Programming ist genau genommen eine Sammlung von emergenten Prozessen, wobei sehr viele, von der Methode her, aus der Prozessoptimierung des Toyota Produktions Systems entnommen sind. Wer sich für diese Denkweisen interessiert, dem sei das Buch „Lean Thinking“ von Womack/Jones, ISBN 3-593-37561-3 oder <http://www.little-idiot.de/teambuilding/EinfuehrungKaizen.pdf> empfohlen.

Führungsstile

Ein Projektleiter/Manager gut beraten, sich in einem Team die psychologisch/menschlichen Eigenschaften seiner Mitarbeiter individuell genau anzusehen, und jeden individuell nach seinen Bedürfnissen zu führen. Gut beraten ist also jemand, wenn er keinen Führungsstil entwickelt, sondern höchst flexibel sich die Freiheit erlaubt, empathisch/einfühlsam sich jedem Mitarbeiter und seinen Stärken/Schwächen individuell zu widmen.

Nach **Steven Reiss** gibt es z.B. zur Orientierung 16 empirisch ermittelte Basismotivationen, welche die intrinsische Motivation eines Mitarbeiters bestimmen. Nichts spricht dagegen, einen Teil der Mitarbeiter nach dem eXterme Programming – Modell arbeiten zu lassen, andere nach agilen Methoden, oder auch wiederum andere Mitarbeiter Pair-Programming oder sogar gänzlich alleine „wurschteln“ zu lassen.

Wer die impliziten Logiken hinter den Modellen verstanden hat, ist in der Lage, sich aus den verschiedensten Softwareentwicklungsmodellen diejenigen Prozesse herauszusuchen, die auf die psychologisch/menschlichen Eigenschaften der Individuen im Team genau passen. Wer Individuen über einen Kamm schert, also allen gemeinsam z.B. XP aufzwingt, riskiert die innere Kündigung von äußerst produktiven und wertvollen Mitarbeitern. Als sehr wirksam hat sich nach meiner Erfahrung erwiesen, das Wissen um die emergenten Prozesse in das Team hineinzutragen, und die Mitarbeiter selber entscheiden zu lassen, wie diese sich organisieren wollen: AP, XP, SCRUM, ...

In der Kunst hat **Louis Sullivan**, bzw. in Deutschland erstmals das **„Bauhaus“** einen Slogan geprägt:

„Form folgt Funktion“ Darf es nicht auch heißen: **„Funktion folgt Form“** ?

Ebenso verhält es sich mit den bestehenden Strukturen innerhalb der Software. Struktur der Zusammenarbeit der Programmierer und Struktur der Software müssen übereinstimmen, unabhängig von den Software-Entwicklungsprozessen, wie man z.B. bei der Analyse der Apache-Software-Struktur sehen kann.

Die Essenz dieser „Alchemie“ - des Wissens über die emergenten Prozesse und deren Wechselwirkung mit den psychologisch/menschlichen Eigenschaften läßt sich im Grunde auch auf andere Bereiche anwenden.

Hat das Team es einmal verstanden, wie man mit wenigen Regeln Arbeits - Prozesse so gestaltet, daß Emergenz auftritt, so führt das dazu, daß Mitarbeiter in einem Team ständig gedanklich auf die Suche nach besseren prozessualen Abläufen gehen, also ständig die Abläufe der täglichen Zusammenarbeit optimieren. Das Management gibt hierzu eigentlich nur die Meta-Prozesse, siehe z.B. CMMI, vor. Die eigentliche Optimierung der Arbeitsprozesse bzw. Programmieraufträge obliegt dann den Mitarbeitern selber. Erfahrungsgemäß steigt die Begeisterung bei den Mitarbeitern stark an – **Als glücklich empfindet sich stets ein Mensch, der eine Vielzahl von Handlungs- und Gestaltungsmöglichkeiten hat.**

Geld ist in unserer Zeit ein Mittel, um mehr Handlungs – und Gestaltungsmöglichkeiten zu haben. Interessanterweise ergeben Umfragen bei hochbezahlten Programmierern, daß Geld nicht glücklich macht, wenn Gestaltungs/Handlungsmöglichkeiten fehlen.

Die OpenSource – Szene beweist, daß Programmierer auch völlig ohne Entlohnung freiwillig in ihrer Freizeit arbeiten, sofern sie die Welt „mit gestalten“ können. Leider verstehen es bislang kaum hochkarätige Manager oder Projektleiter, warum manche niedrig bezahlte Mitarbeiter mit viel Begeisterung mehr leisten, als andere, hochbezahlte Mitarbeiter.

„Im übrigen ist das Prinzip uralte: Das „Rote Kreuz“ „Caritas“, „Bahnhofsmision“, „Verband katholischer Männer/Frauen“ mit oft hunderttausenden, ehrenamtlichen Mitgliedern und Helfern, die unentgeltlich für ein „Ideal“ arbeiten, nutzen schon seit Jahrzehnten dieses emergente Prinzip, welches dann von Richard Stallmann für die Softwareentwicklung unter dem Namen „GNU“ neu erfunden wurde.

Wer einmal die GNU – Prozesse analysiert, findet dort schnell dieselben emergenten Prozesse wieder, die in diesem

Beitrag ansatzweise beschrieben wurden. Nichts spricht daher auch dagegen, für ein kommerzielles Programmier-Projekt - eine riesige Heerschar von freiwilligen Programmierern als Helfer zu gewinnen, um somit die Entwicklungskosten aber vor allem die Kosten für die Fehlersuche gewaltig zu reduzieren. Hier kann vor allem entsprechend dem „**Linus law**“ die Fehlersuche erheblich vereinfacht werden, da diese im Gegensatz zur Programmierung parallelisierbar ist (viele Augen sehen mehr, als wenige!). IBM hat dies schon lange verstanden, siehe das „Eclipse“ und Linux – Projekt.

Eine riesige Schar von professionellen Programmieren (mehrere Mrd. US\$ Volumen jährlich) arbeiten für eine Vielzahl von Open-Source Projekten, allen voran natürlich LINUX, welches komplett alle alten Betriebssysteme von IBM ersetzen soll (OS/2, OS/390, MVS ...) Ob und wie die Intergration funktionieren kann, läßt sich erst nach einer genauen Analyse der Interessen der Open-Source – Community entscheiden. Wer ein noch tieferes Verständnis von Emergenz bzw. emergenten Prozessen in Teams bekommen möchte, dem seien die Kybernetik Bücher von Heinz von Förster „**Der Anfang von Himmel und Erde hat keinen Namen**“ oder meine Übersetzung von SunTse's Kriegsstrategien lesen, welche den Grund für den durchschlagenden Erfolg von alten, chinesischen Generalen oder auch Napoleon anschaulich darstellen, siehe: <http://www.little-idiot.de/teambuilding/SunTsuKunstDesKrieges.pdf>

Der berühmte General von Clausewitz, z.B., der die Grundlagen der Kriegsstrategien der Wehrmacht und der heutigen NATO legte, hat fast 1:1 bei SunTse (SunTse) abgeschrieben. Napoleon hatte die Übersetzung von SunTse's Schrift von einem französischen Mönchen erhalten, der zuvor in China gelebt hatte. „Kunst des Krieges“ ist ein hervorragendes Lehrbuch für Manager / Abteilungsleiter / Projektleiter zur Veränderung von Denkstrukturen, die die Wechselwirkungen der menschliche Psyche mit den prozessualen Abläufen, bzw. die Prinzipien des „Psychodynamischen Prozessdesigns“ verstehen wollen: <http://gutenberg.spiegel.de/clausewz/krieg/inhalt.htm> und <http://www.little-idiot.de/teambuilding/PsychodynamischesProzessdesign.pdf>

Wer selber ein Modell, wie XP z.B. entwickeln oder anpassen möchte, findet unter <http://www.little-idiot.de/teambuilding/PsychodynamischeGesetze.pdf> ein Regelwerk der menschlichen Eigenschaften in Verbindung mit gruppenspezifischen Prozessen, welche es erlauben, Prozesse schon im Vorhinein daraufhin zu untersuchen, ob dieser „laufen wird“, damit dieser nicht immer wieder mangels Motivation der Beteiligten zum Erliegen kommt, immer wieder teuer „angeschubst“ werden muß.

„**Lerne die Regeln, um sie brechen zu können**“ - Miyamoto Musashi

Dieser Beitrag findet sich unter: <http://www.little-idiot.de/teambuilding/EmergenteSoftwareEntwicklungsprozesse.pdf>

„Was **nicht** auf einer *einzigsten Manuskriptseite* zusammengefaßt werden kann, ist **weder durchdacht**, noch **entscheidungsreif**.“ (Dwight David Eisenhower, 34. Präsident der USA 1953-1961; *14.10.1890, † 1969)

Literaturanhang:

1. Bames, B.; Durek, T.; Gaffney, J.: *A Framework and Economic Foundation for Software Reuse*. in IEEE Tutorial Software Reuse Ed. W. Tracz, IEEE Press, L.A., 1988, p. 77
2. Brodie, M.; Stonebraker, M.: *Migrating Legacy Systems*. Morgan Kaufmann Pub., San Francisco, 1995
3. Chapin, N.: *A Measure of Software Complexity*. in Proc. of NCC, Chicago, 1977, p. 995
4. Cimitile, A.; De Lucia, A.; Minro, M.: *A Specification Driven Slicing Process for identifying Reusable Functions*. in Journal of Software Maintenance, Vol. 8, No. 3, May 1996, p. 145
5. Dahl, O.; Dijkstra, E.; Hoare, C.: *Structured Programming*. Academic Press, London, 1972, p. 63
6. Dumke, R.; Foltin, E.; Koeppe, R.; Winkler, A.: *Software Qualität durch Meßtools*. Vieweg Verlag, Braunschweig, 1996
7. Endres, A.: *Software-Wiederverwendung - Ziele, Wege und Erfahrungen*. in Informatik Spektrum, Nov. 1988, p. 85
8. Fenton, N.: *Software Metrics - A rigorous approach*. Chapman & Hall, London, 1991
9. Halstead, M.: *Elements of Software Science*. Elviesier Pub., New York, 1977, p. 79
10. Hehner, E.: *Predicative Programming*. in Comm. of ACM, Vol. 27, No. 2, Feb. 1984, p.134
11. Henry, S.; Kafura, D.: *Software Structure Metrics based on Information Flow*. in IEEE Trans. on S.E. , Vol 7, No. 5, Sept. 1981
12. ISO/IEC: *Software Product Evaluation - Quality Characteristics and Guidelines for their use*. ISO/IEC Standard ISO-9126, 1991
13. Kozacynski, W.; Niny, J.: *Program Concept Recognition and Transformation* in IEEE Trans. on S.E., Vol. 18, No.12, Dec. 1992, p. 1065
14. Lanubile, G.; Visaggio, G.: *Extracting Reusable Functions by Flow Graph-Based Program Slicing*. in IEEE Trans. on S.E., Vol. 23, No. 4, April 1997, p. 246
15. Li, W.: *An Empirical Study of Software Reuse in Reconstructive Maintenance*. in Journal of Software Maintenance, Vol. 9, No. 2, March 1997, p. 69
16. McCabe, T.: *A Complexity Measure*. in IEEE Trans. on S.E., Vol. 7, No. 5, Sept.1981
17. Schach, S.: *Economic Impact of Software Reuse on Maintenance*. in Journal of Software Maintenance, Vol. 6, No. 4, July 1994, p. 185
18. Sneed, H.: *Economics of Software Reengineering*. in Journal of Software Maintenance, Vol. 3, No. 3, Sept. 1991, p. 163
19. Sneed, H.: *Software muß messbar werden*. in Information Management 4/91, Nov. 1991, p.56
20. Sneed, H.: *Understanding Software through Numbers - A metric based Approach to Program Comprehension*. in Journal of Software Maintenance, Vol. 7, No. 6, Nov. 1995, p. 405
21. Sneed, H.: *Encapsulating Legacy Software for Use in Client/Server Systems*. in Proc. of 3rd Conf. in Reverse Eng., IEEE Press, Monterey, Nov. 1996, p. 104
22. Welker, K.; Oman, P.; Atkinson, G.: *Development and Applikation of an automated Source Code Maintainability Index*. in Journal of Software Maintenance, Vol. 9, No. 3, May 1997, p. 127
23. Zuse, H.: *Software Complexity - Measures and Methods*. de Gruyter Verlag, Berlin, 1991
24. Mary B. Chrissis, Mike Konrad, Sandy Shrum: *CMMI. Guidelines for process integration and product improvement*. Addison-Wesley, Boston 2003
25. Ralf Kneuper: *CMMI. Verbesserung von Softwareprozessen mit Capability Maturity Model Integration. 2., überarbeitete und erweiterte Auflage*. dpunkt.verlag, Heidelberg 2006
26. Ernest Wallmüller: *SPI - Software Process Improvement mit CMMI und ISO 15504*. Hanser, München 2006