

OpenSource Entwicklung und ihre Dynamiken

Guido Stepken

OpenSource Software-Entwicklung - Methoden, wechselwirkende, dynamische Prozesse und die psychologischen Hintergründe

Inhaltsverzeichnis

1. OpenSource Software - Entwicklung.....	3
2. Eric Raymond - The Cathedral and the Basaar.....	5
3. Der Mannschaftsgeist.....	7
4. Teamware	8
5. Juristisches.....	8
6. Aspektorientierte Programmierung AOP	9
7. Agile Programming.....	11
8. Extreme Programming.....	12
9. Pair Programming	16
10. Unit Testing	16
11. Graphische User Interfaces und Akzeptanz - Tests.....	17
12. Design Pattern	18
13. Case Tools	20
14. Steven Reiss 16 Lebensmotive.....	20
15. Lösung des Zwergenproblems	22

1. OpenSource Software - Entwicklung

"Deus, dona mihi serenitatem accipere res quae non possum mutare, fortitudinem mutare res quae possum, atque sapientiam differentiam cognoscere.

Dieser Beitrag ist unter nachzulesen.¹

OpenSource ist ein Phänomen, welches vor einigen Jahren nicht denkbar war. Wie kann Software entstehen, ohne daß jemand dafür bezahlt? Der Wert eines Betriebssystems wird mit mehreren Mrd.\$ beziffert, die Wartungskosten (Weiterentwicklung, Optimierung, Treiber, ...) betragen jährlich etwa 40% der ursprünglichen Entwicklungskosten.

Die *"impliziten Logiken"* der Dynamik der Softwareentwicklung in der OpenSource - Gemeinde sind schwierig zu verstehen, da sie durch Wechselwirkungs - Prozesse der menschlichen, psychologischen Eigenschaften und ungeschriebenen Regelwerken der Zusammenarbeit entstehen, die sich selber verstärken bzw. abschwächen. Jemand beginnt, ein Programm für etwas zu schreiben, und flugs gesellen sich hunderte Programmierer freiwillig hinzu, opfern ihre Freizeit und es entsteht - z.B. Linux:

"Unter Millionen Firmen und Programmierern weltweit gibt es immer welche, die ein konkretes, wie auch immer geartetes Interesse an der Weiterentwicklung eines oder mehrerer der unzähligen OpenSource / FreeWare / GNU / BSD - Projekte hat, und in die Weiterentwicklung investiert. Geld spielt dabei immer weniger eine Rolle!"

Das Konzept ist nicht neu: Das rote Kreuz hat z.B. unzählige, ehrenamtliche Helfer und Menschen, die aus purem Idealismus Blut spenden, welches dann für mehrere hundert Euro an Krankenhäuser verkauft wird, welche sich aus unseren Krankenkassen - Beiträgen finanzieren. Man kann auch ohne Bezahlung heutzutage sehr viel bewegen.

OpenSource / FreeWare = GNU Software verwendet die modernsten Erkenntnisse der verteilten Programmierung, die sich hinter Schlagworten, wie Agile Programming, Aspektorientierung, Extreme Programming, Pair Programming, ... verbergen. Die Gemeinde von inzwischen 300.000 Programmierern, über alle Kontinente verteilt, hat recht erfolgreiche Methoden der Zusammenarbeit gefunden, wie Linux, Free/Open/NetBSD - Kernel, die GNU Software, die Benutzeroberflächen GNOME, KDE samt Entwicklungswerkzeugen (Qt, OPIE, GTK+, GTKmm, XPCOM, GCC, GCJ, G++, Python, PERL, RUBY, JAVA, ...), und die immense Zahl von kostenlosen, gut funktionierenden Softwarepaketen zeigen (OpenOffice, GIMP, Firefox, Mozilla, Netscape, ...). Natürlich konnten diese dynamischen Prozesse erst mit Hilfe des Internet in Gang gesetzt werden.

Hierzu muß man die Wechselwirkungen gekoppelter, dynamischer Systeme verstehen lernen, also den Menschen mit seinen psychologischen Eigenschaften, die Struktur der Arbeitsorganisation, die Wurzeln der Motivation. Vielleicht für viele, erfahrene Programmierer nichts neues, aber: *Erfahrung zählt garnichts, man kann auch 25 Jahre alles falsch gemacht haben!*

Die Welt hat sich verändert: Früher hat der Großvater dem Sohn und Enkel erklärt, wie man ein Fahrrad repariert, heute erklärt der Enkel dem Großvater, wie man eine Fernsteuerung programmiert. Ursache - Fortschritt in der Technik. Die inneren Logiken der Welt haben sich verändert, darunter z.B. auch Vertriebslogiken, Herstellungsprozesse, u.s.w. Viele Firmenchefs verstehen einfach nicht, warum und wie sich so grundlegend alles verändert, was jahrzehntelang Gültigkeit hatte - sie geben das Geschäft auf, weil sie impliziten Logiken des NetBusiness z.B. nicht verstanden haben, die nun heutzutage anders sind. Entlassungen sind die Folge. Auf der anderen Seite entstehen riesige Unternehmen, rein virtuell im Internet, wie z.B. einige Software- Entwicklungsfirmen mit mehr als 35.000 Mitarbeitern. Nur derjenige, der diese dynamischen Prozesse versteht, und seine Denkweise ständig anpasst, kommt ohne Probleme und relativ angstfrei durchs Leben. Wer auf der Stelle tritt, muß Niederlagen hinnehmen. Hier nun ein paar interessante Dinge, zur Einleitung in dynamische Denkweisen gedacht:

- Wie kann Software entstehen, ohne daß jemand dafür bezahlt?
- Was war zuerst da? Henne oder das Ei? Ok - anders gefragt: Was war zuerst da, der Künstler oder die Skulptur? Antwort: Die Skulptur. Sie wartete nur darauf, vom Künstler freigelegt zu werden! Abgesehen davon sollte jeder Christ die Frage nach Huhn und Ei eindeutig beantworten können: Und Gott schuf Himmel und Erde, die Pflanzen, die Tiere, die Menschen ... und keine Eier!-
- Es gibt eine *"göttliche Ordnung"*, sagt man. Ein einfaches Naturgesetz besagt, daß ein Stein auf einem Berg solange herunterrollt, bis er - unten am Fuße angekommen - seinen endgültigen Platz erreicht hat, danach bewegt sich nichts mehr. Daraus schließe ich - Was hält die Welt in Gang? - Nichts ist dort, wo es hingehört! Wenn nämlich alles an seinem endgültigen Platz wäre, und die Summe aller Kräfte ausgeglichen wäre, würde sich nichts mehr bewegen. Daraus schließe ich, daß es niemals eine "göttliche Ordnung" gegeben hat, sondern im Gegenteil - Gott hat eine "göttliche Unordnung" geschaffen, und nun ist alles in Bewegung, strebt zu seinem endgültigen Ort.
- In einem fehlerhaften Axiomensystem kann alles bewiesen werden: Angenommen wir definieren in unserem durchweg logischen Zahlensystem eine einzige Ausnahme: $2+2=5$. Daraus folgt dann beliebiger Unsinn, je

nachdem, wie man Argumentiert: $6+6=(5+1)+(5+1)=(2+2)+1+(2+2)+1=10!$ Aus Falschem folgt Beliebiges - "ex falso sequitur quodlibet!". Sprache ist ein Axiomensystem. Durch geschickte Pfadfindung kann in einem fehlerhaften Axiomensystem alles "bewiesen" werden. Verkaufstraining basiert ausschließlich auf geschickter Argumentation mit Suggestion und Empathie, sowie der geschickten Steuerung der Gedankenpfade beim Gegenüber, Gedankenkeime säen, nennt man das. Siehe auch Axiomensysteme, philosophisch/mathematisch betrachtet².

- Was passiert, wenn sich Axiomensysteme durchmengen? Angenommen, zwei Menschen unterhalten sich, oder arbeiten zusammen, der eine beherrscht nur Addition, der anderen nur Subtraktion. Was folgt aus obigem? Angenommen, in Deutschland wird mit Zahlen im Zehnersystem gerechnet, in Luxemburg wird durchgängig im 8er ? System gerechnet. $8+8$ ist in Deutschland $10^1+6 = 16$ $8+8$ ist in Luxemburg jedoch $8^1+8^1=2*8^1 = 20$. Die Vermischung beider (fehlerfreien) Axiomensystemen führt zu absolutem Mist: $(8+8)D-(8+8)L$ ist entweder gleich -4 in Deutschland, oder gleich $+4$ in Luxemburg. Genau dieses Beispiel ist gerade mit den Steuergesetzen zwischen Luxemburg und Deutschland passiert: Die Firma Vodaphone hat durch feindliche Übernahme den Konzern Mannesmann gekauft, das komplette Aktienpaket danach an eine luxemburgische GmbH verkauft, und nach dem Börsenzusammenbruch wieder nach Deutschland zurückverkauft. Der Verlust von 50 Mrd. Euro kann als Verlustvorschreibung in den nächsten Jahren mit den Gewinnen verrechnet werden. Vodaphone muß also in Deutschland die nächsten Jahre keine Steuern zahlen. In Luxemburg sorgen Gesetze im Steuerrecht ebenfalls dafür, daß auch dort keine 50 Mrd. Gewinn anfallen, also ebenfalls keine Steuern zu zahlen sind, da auch in Luxemburg das Aktienpaket ja an Wert verloren hat. Steuergesetze sind jeweils geschlossene Axiomensysteme, wobei bei dereren Vermischung, also internationalen Geschäften die sog. "global player" stets beliebige Möglichkeiten finden, durch geschickte Auswahl des Geltransferpfades (Kurzzeitige Umwandlung in Immobilien, Aktien ...) Steuern zu sparen. Als Notbremse fiel unseren Politikern nur eines ein - die steuerliche Abschreibung für Verlustvorschreibungen zu begrenzen. Mann kann also Verluste aus vergangenen Jahren nur zu $1/3$ mit den Gewinnen verrechnen, damit überhaupt noch Steuern in Deutschland gezahlt werden. Und damit war dann der Standort Deutschland für Investoren noch unattraktiver. Ähnliches bahnt sich bei dem Verkauf z.B. des Kölner Abwassersystems an US-Firmen an, wo US-Anleger den Kauf mit Gewinnen in USA verrechnen können. Die eingesparten Steuern werden dann zu geringen Teilen der Stadt Köln überwiesen. Was als kluger Finanztrick galt, kommt nun als Bummerang auf die Stadt Köln wieder zurück, da die US ? Steuergesetze nun dahingehend geändert wurden, daß so etwas nicht mehr möglich ist. Juristisch ausgebuffte Formulierungen sorgen dafür, daß in diesem Falle u.a. die Stadt Köln alle Kosten des gescheiterten Geschäftes zu tragen hat, und alles rückabgewickelt werden muß. Eine Heerschar von gut ausgebildeten Juristen der US Investoren sind denen des Bürgermeisteramtes offenbar überlegen gewesen.
- Behauptung: Microsoft hat keine Zukunft mehr, angesichts Linux, OpenOffice ... ohne Zukunft - Microsoft weiß das - und hat daher sich für 2.5 Mrd.\$ in die deutsche Telekom eingekauft, welche selber mit Voicestream eine hervorragende Handy - Vernetzung in den U.S.A. bietet, weil die halbierte GSM Frequenz erlaubt, viel weniger Masten aufzustellen, als die Konkurrenz, dies also enorm Kosten spart, wie Ron Sommer damals schon korrekt erkannt hat.
- Gerald Zaltmann (Harvard University - "How customers think") schlug unlängst Mercedes eine Umfrage vor: "Fragt doch mal Eure Kunden, was sie denken, was Mercedes über sie denkt?" - Antwort - Dukatenesel, wenig Qualität für viel zuviel Geld, ADAC Pannenstatistik an Platz 37, Qualität steht in keinem Zusammenhang zum (noch) hohen Image, ... eine sehr negative Prognose, die auf ein gestörtes Hersteller - Kundenverhältnis schließen läßt. Die bisherigen Umfragen der Kundenzufriedenheit sind so mit Suggestivfragen gepflastert, die die Gedanken auf andere Pfade lenken, sodaß kaum ein vernünftiges Ergebnis dabei heraus kommen kann. Und in der Tat muß Mercedes ganze Firmenflotten nach nur wenigen Monaten wieder zurück nehmen, weil die Autos regelmäßig liegenbleiben. Ebenso BMW. Firmenwagen von Audi und Toyota sind qualitativ wesentlich besser.
- Wir sehen nicht, daß wir nicht sehen - Blinde Fleck - Siehe S. 117, "Wahrheit ist die Erfindung eines Lügners", Heinz von Förster. Man male sich in Augenabstand links ein Kreuz und rechts erbsengroß einen Punkt auf ein weißes Blatt. Nun schließe man das linke Auge und schaue mit dem Rechten überkreuzt auf das Kreuz. Wenn man nun den Abstand des Blattes variiert, 10cm-30cm, so verschwindet plötzlich der Punkt. Grund: Dort, wo die Nervenstränge der Netzhaut ins Hirn gehen, können wir nicht sehen, dort sind keine Sehnerven. Wir sehen also nicht, daß wir nicht sehen. Unser Gehirn blendet Bildinformationen aus den Nachbarbereichen ein.
- Zirkuläre Bezüge - "Was ist Sprache?" - Die Frage allein schon ist Anwendung von Sprache - "Um Rekursion zu verstehen, muß man Rekursion verstanden haben!" - Eine Eigenschaft von Universalsprachen, daß man rekursiv Dinge beschreiben kann - analog: "Um Sprache zu verstehen, muß man Sprache verstanden haben!" Liebe - was ist Ursache, was ist Wirkung?
- "Ich sehe, daß Ihr seht, wie ich Euch sehe!" - Menschen verhalten sich komplett anders, wenn sie nicht alleine sind. "candid camera", "Versteckte Kamera" - ein Faszinosum der "Nicht" - Wechselwirkung von Menschen - man wird ja nicht wahrgenommen, kann aber trotzdem zuschauen...

- Entkopplung Gehirn - Umwelt: Es erschallt ein "Miau", und wir errechnen das Bild von Katze vor unserem geistigen Auge - Kognition (siehe Maturana: "Baum der Erkenntnis"). Man schaue sich auf dem Tisch eine Tasse an, schließe die Augen und greife sie. Dies kann nur dadurch funktionieren, daß wir unsere Umwelt vor unserem geistigen Auge "errechnen". Ich folgere daraus: Wir haben keine direkte Wahrnehmung, bzw. benötigen sie auch nicht. Dementsprechend "traumwandeln" wir oft auch durch unsere Welt, ohne z.B. den vollen Mülleimer zu bemerken ;-) - Ein klares Zeichen dafür, daß wir viel mehr "glauben" als "wissen", wenn wir nicht genau hinsehen, bzw. viel mehr kommunizieren müssen. Siehe Heinz von Förster.
- Werbung für Aldi - Kognition (Vorausrichtung einer Denkweise) - Wenn wir dann wirklich etwas brauchen, laufen wir hin, und kaufen es - Autopoiese - siehe Maturana, "Baum der Erkenntnis", bzw. "Was ist erkennen".
- Ist das, was ich erzähle, real? Logikfehler: "Linux existiert!" - Also ist die Erklärung korrekt! Weil die Welt existiert, existiert Gott? Siehe Nicolaus von Autrecourt: Briefe an Pater Arezzo³.

Das Phänomen "OpenSource" ist nicht mit einem statischen Weltbild zu erklären, hierzu werden Erkenntnisse der philosophischen Erkenntnistheorie (Epistemologie), der Kybernetik und der Psychologie benötigt. Der Übergang vom statischen zum dynamischen Weltbild, u.a. auch Abbau von Ängsten durch Unverständnis, Nutzung der Ressourcen für Unternehmen, einen Blick für die dynamischen Prozesse und Wechselwirkungen zu bekommen, ist Ziel dieses Vortrages.

2. Eric Raymond - The Cathedral and the Basaar

Eric Raymond schrieb 1996 ein Text, "The Cathedral and the basaar", der zum ersten Male die Struktur der Entstehung freier Software beschrieb. Der Ursprung jeder freien Software ist ein bisher ungelöstes Problem oder ein unbefriedigtes Bedürfnis. So wurden viele TCP/IP Protokolle, Softwarepakete "erfunden", um ein Problem zu lösen (Streaming Protokolle, PPP, PPPoE,...siehe RFC, IETF.ORG), oder eine Unzulänglichkeit eines bestehenden Softwarepaketes. Oft ist es sogar das mit dem Softwarepaket verbundene Copyright, welches eine Heerschar von Programmierern veranlaßt, hochkomplexe Programme und sogar ganze Betriebssysteme einfach neu zu schreiben. Das von Richard Stallmann begründete Prinzip der GNU Software (Alle Software gehört der Gemeinschaft, jeder darf diese unentgeltlich nutzen, alle Änderungen/Verbesserungen daran sind wieder zu veröffentlichen, Maintainer - Prinzip) hat inzwischen viele Anhänger gefunden. Es ist natürlich nur denkbar geworden durch die globale Vernetzung. Kommerzielle Firmen haben den Zeitgeist erkannt, und gründen ihr Business - Modell ausschließlich auf Dienstleistung, siehe IBM. Jeder lukrative Markt wird inzwischen von GNU Software, bzw. Software, die unter anderen Lizenzen oder auch "Dualen Lizenzen" steht, bedient. Erfolgreiche Ideen werden von der Heerschar von global über 300.000 Programmierern, die ihre Freizeit für ein Ideal opfern, unverzüglich in Code umgesetzt. Kaum ein Softwarepaket, welches nicht nachprogrammiert wurde, und dabei das Original noch an Qualität, Wartbarkeit und Eleganz im Code übertrifft. Siehe www.freshmeat.net⁴ und www.sourceforge.net⁵. Von einem freien Officepaket bis hin zu Raumbelungsprogrammen und ganzen ERP/CRM Paketen ist hier alles zu haben. Eric Raymond beschreibt die wichtigsten Aspekte der Basaar - Methode wie folgt:

- Jedes gute Programm beginnt mit einem ungelösten Problem oder einer Unzulänglichkeit eines bestehenden Programmes, welches einen Programmierer reizt.
- Plane, Code wegzuschmeißen, es wird eh passieren (Fred Brooks: "The Mythical Man Month, Kapitel 11)
- Die Kosten der jährlichen Programmpflege/Wartungskosten liegen bei etwa 40% der Entwicklungskosten. Je mehr User es gibt, desto höher sind auch die Wartungskosten. Mehr User finden mehr Fehler.
- Gute Programmierer wissen, was sie zu tun haben, was sie wie codieren müssen. Großartige Programmierer wissen, was sie neu schreiben müssen, und was sie weiterverwenden können. (Apropos: Man kann in C auch objektorientiert programmieren, wie der C++ nach C - Konverter CFRONT von ATT gezeigt hat!).
- Veröffentliche früh und oft; versuche nicht, fehlerfreie Software auszuliefern, es gibt sie eh nicht. (genutzt wird hierbei die Neugierde und die Experimentierfreude von Anwendern, die Zeit haben, neue Software auszuprobieren).
- Die Wichtigkeit, User zu haben. Höre auf deine Kunden, appelliere an ihren Spieltrieb, baue neue "Features" ein, belohne sie für ihr Feedback.
- Das Gesetz von Linus: Vorausgesetzt, genügend Augen schauen auf den Code, sprich es gibt genug Co-Entwickler oder Beta - Tester, werden alle Probleme in kurzer Zeit bekannt, und die Problemlösung wird schon von irgendwem gefunden werden.
- "Linus's Law": Debugging ist parallelisierbar, weil keine Koordination notwendig ist, im Gegensatz zum Schreiben von Code - Ein wichtiger Hinweis für Entscheider - viele Kunden finden, sofern der Code Open-Source ist, auch die Fehler von alleine. Dies spart enorm Kosten und ist der Hauptgrund für OpenSource!!!

Man beachte auch, daß das UNIT - Testing, ein wesentlicher Aspekt des Extreme - Programming (XP) von den neugierigen Linux - Beta - Testern durchgeführt wird!

- Intelligente Datenstrukturen und simpler Code funktioniert viel besser, als intelligenter Code und primitive Datenstrukturen. Dies ist *das* Argument für OO-Datenbanken, wie z.B. GOODS, ZODB, PostgreSQL (Kern ist OO!).
- Behandle Beta - Tester als wären sie deine wertvollste Resource, und sie werden zu genau dieser. Wertschätzung ist ein wichtiges, psychologisches Kriterium der Motivation und ein Steuerungselement, Menschen für etwas zu begeistern.
- Gute Ideen zu haben, ist nur die zweitbeste Möglichkeit. Gute Ideen von Usern aufnehmen, ist die beste Idee.
- Die besten und innovativsten Ideen und Fortschritte beim Programmieren macht man, wenn man erkennt, was am Konzept falsch war, und es einfach - gnadenlos und ohne mit der Wimper zu zucken, über Bord wirft. Sei immer bereit, Code einfach wegzuworfen, Du wirst es eh tun, wenn das Konzept nicht stimmt, allerdings bis dahin viel Zeit verschwenden.
- Schaffe möglichst viele Schnittstellen für Plugins, dokumentiere sie perfekt, animiere, Module zu schreiben, siehe Linux Modules Plugin, Apache Module (PHP, PERL, JAVA...), belohne Programmierer, indem sie auf der Projekt - Homepage einen "Ehrenplatz" bekommen!
- Dokumentiere besonders die Schnittstellen für Erweiterungen besonders gut, schreibe Tutorials, Howtos, kleine, einfache Anleitungen, die das Design der Software erklären. Du wirst so viele Mitarbeiter finden.

Entscheidend für den Erfolg von freier Software ist, ob ein "Basar-Projektleiter" Menschen für ein Projekt begeistern kann. Kann er dies, so wird er mehr Manpower für das Projekt zur Verfügung haben, als jeder kommerzielle Softwarehersteller (OpenOffice, Firefox, Netscape, Mozilla). Folgende Eigenschaften sind ebenfalls wichtig:

- Er muß Leute davon überzeugen, daß das Softwarepaket sich in naher Zukunft gut entwickeln wird, Meilensteine nennen, Features ankündigen, das öffentliche Interesse in Zeitungen und Fachzeitschriften wecken, um möglichst viele Co-Entwickler für das Projekt begeistern zu können (Ein Apell an Menschen mit leichter Profilneurose, siehe Belohnungssysteme) Bill Gates hat in einem Interview gesagt, daß Microsoft noch nie deswegen ein neues Softwarerelease verkauft hätte, weil es weniger Fehler enthielt, sondern immer nur dadurch, daß es neue Features enthielt (Neugierde ist die Vorfreude auf Erkenntnis).
- Ein guter Koordinator muß kein brillianter Programmierer bzw. Softwaredesigner sein, er muß aber erkennen können, wann jemand anderes tatsächlich eine gute Idee hat, die das Team weiter bringt, wie z.B. die Reduktion der Komplexität, der Code - Abhängigkeit untereinander.
- Ein guter Projektleiter versteht es, die Aufmerksamkeit auf spezielle Aufgaben zu lenken: Aus der Sicht des Programmierers schaut dies dann so aus: Wenn Du die richtige Einstellung hast, finden die interessanten Probleme Dich! (eine "teleologische" Erklärung, vom Ziel her begründet, siehe philosophische Erkenntnistheorie = Epistemologie)
- Wenn das Interesse an einem Programm nicht mehr besteht, ist es die letzte Aufgabe eines Basar-Projektleiters, es in fachkundige Hände weiterzureichen

Diese Punkte laufen im Endeffekt auf eines hinaus - Programmieren nicht selber, entwickle niemals selber Software, spare Geld und Entwicklungskosten, lasse Software von der OpenSource - Gemeinde entwickeln (was nur funktioniert, wenn ein breites Interesse da ist):

- Redhat hat alle Entwicklung an die OpenSource Gemeinde übergeben: Projekt Fedora⁶.
- Netscape hat die Entwicklung eingestellt und dies als Mozilla Projekt⁷ veröffentlicht. Mit Hilfe der Crossplattform - Library XPCOM wurden u.a. neben Firefox auch OpenOffice entwickelt.
- OpenOffice⁸ ist wohl neben Linux und GNU Tools das komplexeste Projekt. Es ist ebenfalls mit XPCOM programmiert.
- Compiere CRM/ERP Tool⁹ wird bald Ersatz für SAP R/3.
- PostgreSQL¹⁰ entwickelt sich zu dem besten Datenbank - Server überhaupt - Oracle - Kompatibilität ist fast erreicht.
- Firebird als Oracle Ersatz (Fyracle)¹¹
- PHPNuke Portalsoftware¹² wurde mehrfach neu geschrieben, mangels Designfehlern (mal fehlte ein Language - File, dann Skins)
- Postnuke¹³ - ein von PHPNuke abgespaltenes Projekt, mangels Designfehler des damaligen PHPNuke.

- Zope CMS - System¹⁴ Zope ist eines der modernsten Softwareprojekte überhaupt: XP Development, UNIT Tests, ...
- OpenCMS¹⁵.
- Softwareentwicklung unter Linux¹⁶

Es gibt aber auch eine ganze Reihe erfolgloser Projekte, wie z.B. Netscape¹⁷ (Code nicht mehr wartbar), CA Ingres¹⁸ (zu spät, PostgreSQL und MySQL machen das Rennen), SAPDB - nun MAX-DB¹⁹ (zu spät, PostgreSQL ist besser), unzählige Linux Distributionen, Datenbankhersteller (Gupta), Novell, Content Management Systeme, ...

3. Der Mannschaftsgeist

Aus: Ryle, Der Begriff des Geistes. übers. von: Kurt Baier, 1. Aufl., Stuttgart: 1987, Reclam, S. 15: *"Ein Südseeinsulaner sieht seinem ersten Fußballspiel zu. Man erklärt ihm die Funktion des Torwarts, der Stürmer, der Verteidiger, des Schiedsrichters usw. Nach einer Weile sagt er: "Aber da ist doch niemand, der den berühmten Mannschaftsgeist beisteuert. Ich sehe wer angreift, wer verteidigt, wer die Verbindung herstellt usw.: aber wessen Rolle ist es, den Mannschaftsgeist zu liefern?"*.

Das Wort "Mannschaftsgeist" ist eigentlich ein typischer Kategorienfehler, sprich falsche Verwendung des Wortes "Geist" in diesem Zusammenhang (man achte auf die sprachlichen Kategorienfehler in der Psycho / Eso / Para / Meta/ -Szene, wo die Leute glauben, daß die wilde Vermischung von Worten, die kategorienmäßig nicht zusammengehören, bereits eine neue Erkenntnis darstellen, und bewußtseinerweiternd wären, tatsächlich erfolgt nur Endorphinausschüttung, Glückshormone). Maskottchen, Firmen - Logos sind Symbole des Mannschaftsgeistes, des Zusammenspiels eines Teams. Hierbei vermischen sich psychologische Eigenschaften, Verhaltenslogiken des Systems "Mensch", mit den Logiken des Fußballs, also einem weiteren Regelwerk, auch Axiomensystem genannt. Aus dieser Wechselwirkung entstehen beobachtbare, weitere Regelwerke, wie z.B. Mann/Raumdeckung, Abseitsfalle, u.s.w. Es sind Regelwerke 2. Ordnung, die aus dem dynamischen Zusammenspiel schwach gekoppelter Regelwerke 1. Ordnung von Systemen entstehen. Z.B. ist die Bewegung eines Pendels einfach kalkulierbar, jedoch diejenige eines Doppelpendels bereits chaotisch. "Implizite Logiken" bezeichnen diejenigen Regelwerke, die nicht definiert wurden, und scheinbar aus dem Nichts zu entstehen scheinen. Beispiele: Wo steht geschrieben, daß erst alle aus der Straßenbahn aussteigen müssen, bevor man selber einsteigt? Manndeckung / Raumdeckung / Abseitsfalle im Fußball? Jede Art Strategie ist ein Regelwerk 2. Ordnung, weil es die Art der dynamischen Wechselwirkungen beschreibt. Ein Pendel ist physikalisch einfach zu beschreiben. Ein Doppelpendel jedoch zeigt völlig chaotisches, unkalkulierbares Verhalten, typisch für stark gekoppelte Systeme. Auch bei Menschen ist dies zu beobachten - bei Streit, Streß gehen sich Menschen aus dem Weg - Entkopplung findet statt, bevor eine Situation chaotisch eskaliert, sich die Dinge aufschaukeln, ähnlich einer Resonanzkatastrophe, die sich z.B. bei einer schwingenden Schaufensterscheibe anbahnt, und bis zur Explosion führt (man kann diese mit viel Gefühl und Übung mit einem kleinen Finger zum Bersten bringen).

Andererseits symbolisiert ist Mannschaftsgeist den Willen zum gemeinsamen Sieg, oft dargestellt durch Firmenlogos, Maskottchen, Flaggen, Fahnen, Abzeichen, u.s.w. Wie die Dynamiken zwischen uns Menschen ablaufen, was z.B. Kognition, Autopoiese ist, zeigen die überaus interessanten Bücher der Kybernetiker Heinz von Förster, "Sicht und Einsicht", "Wahrheit ist die Erfindung eines Lügners", "Der Anfang von Himmel und Erde hat keinen Namen" und Maturana "Baum der Erkenntnis". Sie beschreiben die Wechselwirkungen (schwach) gekoppelter Systeme und deren Dynamiken, die sich daraus ergeben. Siehe auch Little-Idiot Teambuilding²⁰.

Wie blockiert wir Menschen sind, wenn es um die Lösung einfachster Probleme geht, zeigt dieses Rätsel:

Zwerge im Wald stehen früh auf, greifen im Dunkeln beim Verlassen ihrer Höhle blind eine Mütze (rot oder grün), die sie den ganzen Tag aufhaben, ohne zu wissen, welche Farbe diese hat. Zwerge arbeiten harmonisch den ganzen Tag, ohne zu kommunizieren, jeder hat seine genau definierten Aufgaben. Nun kommt ein Zwerg daher, vermittelt, daß diese sich auf einer Linie aufstellen sollen, sortiert nach roten und grünen Mützen. Ohne daß die Zwerge kommunizieren und ohne daß jemand diese einteilt, stellen sie sich in einer Linie auf, perfekt sortiert nach roten und grünen Mützen. Wie machen sie das? Welche impliziten Logiken stecken dahinter, welche sind 1. Ordnung, welche 2. Ordnung (die sich erst aus der Dynamik des Zusammenspiels ergeben). Die Lösung ist nicht einfach - siehe Abschnitt 15. Wie sieht die Lösung mit 3 verschiedenen Mützenfarben aus?

Ein weiteres Rätsel verbirgt sich hinter dem HTTPS - Protokoll, welches einen angeblich sicheren Krypto-Tunnel z.B. zur Bank aufbaut. Wie kann das, wo doch keinerlei Passworte ausgetauscht werden. Normalerweise benötigen ja sowohl Sender, als auch Empfänger zumindest ein gemeinsames Passwort, welches Dritten unbekannt ist, ein sog. "shared secret" also, welches über andere Wege mitgeteilt werden muß, oder zuvor mündlich vereinbart wird. Wie kann man Informationen austauschen, ohne daß ein "Man in the middle" mitlauschen kann, und ohne daß dieser die Passworte abhören kann? Das Rätsel ist einfach zu lösen, wenn man sich folgendes Beispiel anschaut: A und B möchten geheim Briefe austauschen, die niemand mitlesen kann. Hierzu verwenden

den sie eine solide Truhe, an welcher zwei Vorhängeschlösser angebracht sind. A tut nun eine Nachricht in die Truhe schließt mit seinem Schloß ab, behält aber den Schlüssel für sich. Er sendet die Kiste nun B, mit der Bitte, die Truhe mit einem zweiten Vorhängeschloß zu verschließen, ebenfalls den Schlüssel zu behalten, und die Truhe wieder zurückzusenden. A entfernt nun sein Schloß von der Truhe, und sendet diese wieder an B. Dieser kann nun mit seinem Schlüssel die Truhe öffnen, und die Nachricht von A lesen. Zu keiner Zeit war also die Truhe unverschlossen unterwegs, und zu keiner Zeit mußte irgendein Schlüssel übermittelt werden.

Dieses geniale Verfahren gehört zur Gattung der "Zero Knowledge Probleme", und wurde von Whitfield Diffie und Martin Hellmann, siehe US Patent # 4,218,582 (<http://www.uspto.gov>), 1980 als "Public Key Cryptographic Apparatus and Method" bekannt, oder auch unter "Diffie-Hellmann key exchange". Es ist Basis der SSLv2 Verschlüsselung in Browsern. Und nun die Frage: Wie kann es überlistet werden? Die Lösung ist recht einfach...

Ein weiteres, interessantes Phänomen ist ein Buffet. Man stelle sich vor, ein Buffet wurde gerade vom Gastgeber eröffnet, und schon bildet sich eine lange Schlange, obwohl - wenn alle parallel an das Buffet gehen würden, die Plätze schnell tauschten, alle viel schneller zu ihrem Essen kommen würden. Welche implizite Logiken hindern Menschen daran, dies so zu tun? Welchen persönlichen Vorteil hat Mensch davon, sich trotzdem in der Schlange anzustellen? Oder sind wir alle nur nicht in Problemlösungsstrategien ausreichend geschult worden?

4. Teamware

Softwarepakete für Programmierer - Teams gibt es viele. Hier nun diejenigen, die gerne und mit viel Erfolg eingesetzt werden:

- Forum PHPBB Forum²¹
- Trouble Ticket System OTRS²²
- IRC - Developer - Channel FreeNode.net²³
- Software Dokumentationswerkzeug Doxygen²⁴, Beispiel Netscape Dokumentation²⁵. Man beachte hierbei, daß Doxygen die Diagramme vollautomatisch aus dem Quellcode erzeugt, insbesondere die *collaboration diagrams*. Ohne DoxyGen und seine überragende Fähigkeit, Codestrukturen zu visualisieren ist ein Programmieren im Team völlig unmöglich, geschweige denn daß ein neuer Programmierer nach kurzer Zeit sich einarbeiten und produktiv mitarbeiten kann.
- Editor, Design-, Refactoring-Werkzeug Eclipse²⁶, der leistungsfähigste Editor überhaupt, dank seiner unzahl von äußerst leistungsfähigen Plugins
- BUGZILLA²⁷. Ohne Bugtracking - Systeme, intern sowie extern für User sollte kein Programm mehr entwickelt werden. Beschwerden, Bugs gehen nur allzugerne unter im Alltag.
- Alle Mailing-Listen im Überblick²⁸
- Google Web²⁹, Google Groups³⁰.
- www.bugzilla.org³¹.
- EGroupware.org³². Für die tägliche Koordination der Arbeit, Projektmanagement, Todo-Listen, die Zurückverfolgung von Mail-Dialogen, Mailing - Listen zur Information der Teams ist eine vernünftige Teamware notwendig. Egroupware leistet das.
- PHPNuke³³ und Postnuke³⁴.
- Ankündigungen für OpenSource/Freeware: Freshmeat.net³⁵, Sourceforge.net³⁶
- CVS Home³⁷ + CVS Addon's³⁸. CVS ist, trotz seiner Addons ein recht rückständiges Programm, welches viele wichtige Features nicht enthält, die sich jedoch in Bitkeeper wiederfinden:
- Bitkeeper³⁹ - Ein CVS - System, jedoch sehr viel besser, als CVS, siehe unter "Comparisons".
- Linux Solution Guide⁴⁰

Man beachte, daß der alleinige Einsatz von Teamware bzw. Groupware kein Team zusammenschweißen kann, wenn nicht zuvor schon Dynamiken installiert / in Gang gesetzt wurden, daß die Leute von alleine miteinander kommunizieren wollen.

5. Juristisches

Juristische Dinge sind hochkomplex, besonders im internationalen Recht. Daher nur ein grober Überblick über die Charakteristika der verschiedenen Modelle:

- GPL - GNU Public License - Wer mit Software programmiert, die unter GPL veröffentlicht wurde, muß diese Software selber wieder unter GPL stellen, dazu gehören auch Module, nicht aber Programme, die eigenständig installiert werden, und nur via Port, UNIX Socket oder IPC verbunden sind. Aller Code muß veröffentlicht werden. Microsoft nennt dieses "viral". Alle Änderungen am Code sind an den Maintainer zurück zu melden, sofern das Programm veröffentlicht wird. Was eine Firma intern oder eine Privatperson programmiert, muß nicht veröffentlicht werden.
- LGPL - Wer LGPL Code verwendet, darf beliebig auch kommerzielle Programme damit herstellen, ohne Code zu veröffentlichen.
- BSD License - Jeder darf mit dem Code machen, was er will, Namen verändern, u.s.w. Sehr viel BSD Code ist in Windows XP enthalten, Mac OS X besteht fast vollständig aus BSD - Code (Darwin ist FreeBSD!!!).
- X11, Apache, Apple (Darwin)... Lizenzen - jeder darf mit dem Code machen, was er will, auch kommerzielle Programme schreiben. Für den Fall, daß patentrechtliche Probleme auftreten, kann der Code zurückgezogen werden.
- Duale Lizenzen, wie z.B. bei MySQL haben hauptsächlich zwei Gründe. Erstens möchte MySQL AB bei dem kommerziellen Einsatz mitverdienen, was bedeutet, daß eine Firma, die kommerzielle Anwendungen programmiert, und die Mysql-Client Libraries mit in ihr Programm einkompiliert, Lizenzgebühren zahlen muß, aus denen MySQL weiter entwickelt wird. Zum Zweiten hat Microsoft untersagt, daß GPL Software auf Windows XP installiert wird. Nach internationalem Recht kann der Autor der Software darüber bestimmen, wofür seine Software eingesetzt werden darf, z.B. nicht für Kriegszwecke. Damit MySQL auf Windows XP eingesetzt werden darf, muß die kommerzielle Version erworben werden. Dennoch darf, dank der GPL, MySQL auch für kommerzielle Zwecke eingesetzt werden, sofern kein Code von MySQL in dem kommerziellen Programm eingebaut wurde, also z.B. MySQL auf CDROM beiliegend, mitgeliefert wird.
- SuSE / Novell Linux z.B. unterliegt besonderen Rechten. Niemand darf z.B. YaST, das Installations - und Administrationswerkzeug für eine eigene Distribution verwenden, wie dies z.B. mit RedHats Anaconda Werkzeug möglich ist, welches in Knoppix eingesetzt wird (anaconda , kudzu). Die jeweils neueste SuSE - Distribution findet sich daher nie im Internet zum kostenlosen Download.

Linux besitzt zwei grafische Benutzeroberflächen, von denen beide auf Windows portiert wurden, jedoch nur eine wirklich frei ist - GNOME. KDE insbesondere die Programmierung mit Qt ist Lizenzgebühren - pflichtig. Wer plattformunabhängig und lizenzfrei programmieren möchte, sollte sich daher OPIE, FTK, XUL, XPCOM anschauen, mehr hierzu siehe <http://www.little-idiot.de/linuxsolutionguide/>

6. Aspektorientierte Programmierung AOP

Aspektorientierung zu verstehen, ist recht abstrakt. Am besten trifft der Ausdruck *verweben* von Klassen zu. Hierbei wird eine bestehende Klasse mit einer anderen so verwoben, daß diese sich, abweichend von ihrem ursprünglichem Entwurf, völlig anders verhält. Das Besondere hierbei ist, daß kein Eingriff in den Code erforderlich ist. Dies ist *code reuse* par excellence. Ein Beispiel zeigt mustergültig die Weiterverwendung des Codes einer bestehenden Klasse, also *code reuse* in seiner elementarsten Form, ein Beispiel in der Programmiersprache Python:

```
import aspekts.py

class C:
    def machwas(self):
        print "machwas"

def aspekt(self, *args, **keyw):
    print "vorherdies"
    rv = self.__proceed(*args, **keyw)
    print "nachherdas"
    return rv
```

Die Klasse C, ohne diese zu verändern, wird erweitert:

```
wrap_around( C.machwas, aspekt )
```

Nun führen wir aus:

```
o = C()  
o.machwas()
```

Ausgabe:

```
vorherdies  
machwas  
nachherdas
```

Wie man sehen kann, werden hier zwei Codes miteinander *verwoben* (*code weaving*). Zwei Anwendungen für AOP sind z.B. Logging, Persistenz (Hibernate) und Profiling (Zeitmessungen). Während man ohne AOP in jeder Klasse Logging - Funktionen bzw. Profiling - Funktionen implementieren muß, kann man mit AOP die Klassen so belassen, wie sie sind, sie behalten also ihre ursprüngliche Aufgabe. Logging bzw. Profiling wird über Aspekte eingeführt. Nur eine Klasse enthält Code für Logging bzw. Profiling, was die Wartbarkeit des Codes sehr vereinfacht. AOP hilft sehr bei folgenden Zielen in der Teamprogrammierung:

- Klassen sollten nie mehr, als eine Aufgabe erfüllen - Bessere Trennung der Zuständigkeiten von Modulen *separation of concerns*
- Quellcode wird übersichtlich gehalten - Klareres Design
- Sehr gute Wiederverwendbarkeit von Code - Weniger redundanter Code
- Bei Änderungen muß nur sehr wenig Code verändert werden
- Erheblich bessere Wiederverwendung von Modulen - Hohe Modularität (*tangled code*)
- Bei Änderungen der Anforderungen muß nicht an vielen Stellen Code verändert werden, sondern nur in einer Klasse zentral, welches sich dann auf viele Klassen auswirkt.
- Einfachere Fehlersuche - Fehler haben eine hohe Lokalität, Korrekturen nur in äußerst wenig Code-Teilen notwendig. Die in Java oft in den Klassen behandelten Exceptions machen bei normaler Programmierung 11% des Codes aus, bei AOP nur ca. 3%

Dies führt insgesamt zu besserer Qualität von Software und großen Produktivitätssteigerungen bei deren Entwicklung, weil die "innere Reibung" erheblich reduziert wird. Wann sollte man AOP einsetzen? Hierzu gibt es eine ganz klare Antwort: Immer dann, wenn unklar ist, wo, in welcher Klasse etwas untergebracht werden soll, wie z.B. Logging. Wenn man sich z.B. das Apache Tomcat Modul anschaut, so findet sich "logging" über hunderte von Klassen verteilt - eine sehr schlechte Modularität, während hingegen das Modul "UML pattern matching" nur in zwei Klassen gekapselt ist.

AOP erweitert die Möglichkeiten der Strukturierung von Code:

- Prozedurale Programmierung erlaubt die Strukturierung von Code nur in einer Dimension: Nur GoSUBS und GOTOs ermöglichen eine geringe Strukturierung.
- Objektorientierte Programmierung (OOP) erweitert die prozedurale Programmierung um Objekte, die ihre Eigenschaften vererben können, was eine Schachtelung des Codes ermöglicht, jedoch auch hohe Abhängigkeiten schafft, was die Entwicklung gerade bei komplexen Verflechtungen des Codes in großen Projekten oft stagnieren läßt. Außerdem wirken sich kleine Fehler an den Basisklassen norm auf alle weitere Klassen aus. Sie potenzieren sich.
- Aspektorientierte Programmierung (AOP) erweitert OOP wiederum um eine Dimension, nämlich die Aspekte, die den oft hochgradig verschachtelten Code entzerren, und die Code-Teile, die viele Klassen gemeinsam haben, in eigene Klassen auslagert. Dies ermöglicht eine hohe Modularität des Codes und vermeidet insbesondere doppelten Code.

Die Definition von AOP ist schwierig, hunderte Bücher wurden geschrieben (von denen kaum eines leicht verständlich geschrieben ist):

- Aspekte sind Aufgaben, die nicht klar Objekten zugeordnet werden können.
- Aspekte können getrennt vom Objekt programmiert und bearbeitet werden.
- Der Objektcode ist unabhängig vom Aspektcode
- Aspekte und Objekte werden miteinander verwoben um ein gewünschtes, verändertes Verhalten eines Objektes zu erreichen.

- Das Verweben kann auch mit Hilfe von Markern (Markerungspunkten) komplexer gestaltet werden.
- Aspekte können vererbt werden

Folgende Beispiele für Python, Java und C++ zeigen eindrucksvoll die Möglichkeiten, die AOP bietet:

- Python aspektorientierte Programmierung⁴²
- Python Lightweight AOP⁴³
- Python LogiLab AOP GPL Modul⁴⁴
- AspektJ JAVA AOP Implementierung mit Eclipse⁴⁵.

Der Ursprung von AOP in JAVA (AspektJ) liegt 1997 beim Xerox Palo Alto Research Center. AspektJ ermöglicht es, Aspekte unter JAVA zu beschreiben, sodaß ein eigener Compiler die Aspekte in den Code einwebt. Es ist heute fester Bestandteil des IBM Eclipse - Projektes, aber auch in JBuilder enthalten.

Wann also sollte man AOP einsetzen? Martin Lippert und Cristina Videira Lopes haben in "A Study on Exception Detection and Handling using Aspekt Oriented Programming" einige Fälle untersucht: Weitere Dokumente zu AOP im Internet⁴⁶. AOP unter JAVA bietet nicht in jedem Fall elementare Vorteile gegenüber OOP bzw. herkömmlicher Programmierung, insbesondere dann nicht, wenn die *Denkweisen* nicht geschult wurden. "Lightweight AOP" - Module gibt es für jede OOP, deren Nutzung hat sich als nützlich herausgestellt, und ist leicht im Team einzuführen.

7. Agile Programming

Agile Programming = AP ist ein Schlagwort, hinter welchem sich Schwerpunkte auf bestimmte Methoden der Softwareentwicklung verbergen:

- Individuen und Interaktionen anstelle von aufwändigen Entwicklungs - Prozessen (die typischen Kaffee-Kränzchen und teuren Werkzeugen, deren Eigenschaften sich nicht entfalten können).
- Funktionierende Software anstelle umfangreicher Dokumentationen, Powerpoint - Blendwerken.
- Mitarbeit des Kunden anstelle riesiger Vertragswerke.
- Flexibles Antworten auf Veränderungen anstelle einem Plan zu folgen.

Hiermit sei nicht gemeint, daß Pläne, Vertragswerke, Dokumentationen und Werkzeuge nicht nötig wären, sondern vielmehr, daß der Schwerpunkt eher auf der anderen Seite (Seite funktionierender Codes) angesiedelt sind. Hieraus ergeben sich weitere Aspekte:

- Höchste Priorität haben die Kundenanforderungen durch frühe Veröffentlichung von Versionen zum Testen.
- Änderungswünsche sind stets willkommen, auch gegen Ende der Softwareentwicklung. "Agile Softwareentwicklung" hat stets das Softwaredesign im Auge, zugunsten späterer Weiterentwicklung, wie *code reuse*, *refactoring*, *Wartbarkeit*, ...
- In kurzen Abständen werden stabile Versionen herausgebracht, um frühzeitig Fehler im Design entdecken zu können, besonders der GUI.
- Anwender und Entwickler müssen fast täglich zusammenarbeiten.
- Projekte (=Aufgabe für eine Gruppe von Menschen) werden immer nur hochmotivierten Personen anvertraut (weniger motivierte Personen raus, sie rauben den anderen nur Zeit und Begeisterung!!) Man gibt ihnen die Umgebung und Unterstützung, die sie benötigen, und vertraut ihnen, daß sie die Aufgabe lösen.
- Die effektivste Methode, Informationen im Team zu verbreiten, ist - Kommunikation - Angesicht zu Angesicht.
- Funktionierende Software ist das Maß aller Dinge.
- *Agile Programming* basiert auf kontinuierlicher Entwicklung. Auftraggeber, Entwickler und Anwender werden gleichermaßen in den Prozess einbezogen.
- Dauerhaftes Streben nach technischer Brillanz und gutem Design sind Anforderungen, die fester Bestandteil von AP sind.

- Einfachheit - die Kunst, mit wenigen Codezeilen maximale Wirkung zu erzielen, also höchste Brillanz ist Maxime von AP. Die Arbeit macht als Programmierer dann sehr viel mehr Freude, Endorphin- Ausschüttung schon beim Code - Lesen.
- Die besten Architekturen, Software - Designs, Pattern, entstehen aus der Selbstorganisation des Teams, wobei man jedoch immer beachten sollte, daß man sehr viel Zeit sparen kann, wenn man Strukturen aus ähnlichen OpenSource - Projekten sich genau anschaut, z.B. mit DoxyGen.
- Teamleiter halten sich strikt aus der Entwicklung der Design - Patterns heraus, sie haben eh andere Aufgaben.
- Regelmäßig trifft sich das Team, um über effizientere Methoden und Designs kritisch nachzudenken, und die eigenen Arbeits/Verhaltensweisen anzupassen.
- Sobald ein effizienterer Weg entdeckt ist, wird dieser auch genutzt. Es wird sich später bei der Wartbarkeit der Software enorm auszahlen.

8. Extreme Programming

Extreme Programming XP beinhaltet alle Aspekte des Abschnitt 7, bietet den Teams darüber hinaus aber weiter differenzierte Methoden für die vier sich bis zur endgültigen Fertigstellung der Software ständig wiederholenden Entwicklungs-Zyklen (Iterationen) an:

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

Siehe auch www.extremeprogramming.org⁴⁷. 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 gemittelt nur ganz 5! Codezeilen je Arbeitstag. *Nichts ist beständiger, als die Veränderung* - XP ist ein bewußter Prozess von bewußtem 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 Boeing erfundenem "KVP" (Kontinuierlicher Verbesserungs - Prozess), der zur Sicherung der Qualität im Fertigungsprozess bei Flugzeugen, aber auch in der japanischen Autoindustrie und sehr erfolgreich bei Porsche angewendet wird. Mehr hierzu siehe <http://www.little-idiot.de/teambuilding/kaizen.pdf>⁴⁸. 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 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 einen Teammitglied nur schwer zu verstehen, daß Prozesse, bei welchen jedes Teammitglied individuelle Nachteile hat, diese für das Team insgesamt jedoch von Vorteil sind:

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.
- 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.
- Die Geschwindigkeit der Projektentwicklung wird in jedem Entwicklungs-Zyklus erneut abgeschätzt.
- Die Entwicklungszyklen werden alle 1-3 Wochen, aufgrund der sich ständig ändernden Anforderungen und Kundenwünsche, wieder und wieder erneut definiert.
- 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.
- 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, siehe Abschnitt 9, 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.
- 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.

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 Bezeichnern für Code - Abschnitte oder Programmen/Unterprogrammen erleichtert die Kommunikation im Team, siehe auch Abschnitt 12.
- 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!, DoxyGen geht aber auch). Jede Karte repräsentiert ein Objekt mit Abhängigkeiten, zu 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. Hierbei werden in kurzer Zeit durch das gemeinschaftliche Denken Schwächen im Design entdeckt und korrigiert. Existieren zuviele Karten und Klassen, so wird die Zahl auf wenige je Person begrenzt.
- 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, sind die *"Number of Lines Not Written"*. Im Design jedoch berücksichtige stets die Möglichkeit, diese Funktionalitäten später hinzuzufügen zu **können**. Konzentriere Dich auf die morgendlichen Besprechungen und dein Tagespensum.
- 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, siehe Abschnitt 6. 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", siehe Abschnitt 12.
- 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. Mehr hierzu siehe Abschnitt 10.
- 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 Gesprächs 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 entgeltigen 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*. 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 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>. 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.
- Optimize niemals während der Entwicklungsphase. Erst muß Code funktionsfähig sein, später dann können Nadelöhre beseitigt werden. Versuche niemals, zu erraten, wie groß die Verbesserung der Performance sein könnte, sondern messe sie.
- 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 Programmiertätigkeit ein schöpferischer Akt ist, und ein Programmierer sehr viel mehr nachdenkt, als tatsächlich am Computer Code eingibt. 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 wegfallen läßt. XP ist so konzipiert, daß ständig Planung, Design, Codieren und Testen in kleinen Entwicklungs-Zyklen interaktiv im Wechsel stattfindet. Hierdurch werden frühzeitig unrealistische Ziele erkannt und korrigiert, oder es werden andere Wege gefunden. Oft nämlich ist es einfacher, Arbeitsabläufe 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, welcher Tests hohe, und welche niedrige Priorität haben. Eine *user story* gilt als nicht vollständig, wenn diese nicht alle Akzeptanz - Tests bestanden hat. das 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. Mehr zu Akzeptanz - Tests siehe Abschnitt 11.

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 selbstkorrigierenden, dynamischen Prozesse eigentlich funktionieren, welche impliziten Logiken hinter jedem einzelnen Punkt verborgen sind. Gelegentlich 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, sprich Endorphine freisetzt. 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 werden. ZopeX3, z.B. verwendet Unit - Testing, siehe Abschnitt 10.

9. Pair Programming

Pair - Programming ist für fast alle Programmierer sehr ungewohnt. Als Einzelkämpfer am Code ist der Gedanke, daß jemand dazwischen pfuscht, unerträglich. Man teilt sich, im ständigen Wechsel programmierend, nämlich einen PC. Hierzu muß zunächst aufwändig in dem Kopf jedes einzelnen ein Umdenkprozeß stattfinden, bevor überhaupt jemand die enormen Vorteile für das Gesamtteam erkennen kann, sprich - man muß es zusammen geübt haben, und sich persönlich von den Vorteilen überzeugt haben. Es ist für eine Einzelperson nicht zu verstehen, daß persönliche Nachteile beim Programmieren sich in der Gruppe als Vorteil erweisen können.

Programmieren ist ein schöpferischer Prozess, in welchem jeder Mensch 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". Ähnlich, wie wir nicht auf die algorithmische Lösung des Zwergenproblems kommen, findet man alleine fast nie die optimale Lösung. (Beispiel: Der Suchbaum des Spiels "Nimm" läßt sich stark vereinfachen! Zwei Spieler nehmen aus einer Reihe von 10-30 Hölzchen abwechselnd 1-3 Hölzchen. Wer das letzte H. nehmen muß, hat verloren. Es muß theoretisch ein Suchbaum mit 3^{30} Möglichkeiten aufgestellt werden...es geht aber auch viel, viel einfacher!). Die besten Lösungen, die effektivsten Algorithmen und den kürzesten Code findet man immer durch offene Kommunikation und Diskussion. Pair - Programming erzwingt dies. Da viel weniger Code generiert wird, dieser viel besser durchdacht ist, viel weniger Code wieder verworfen werden bzw. überarbeitet werden muß, ist Pair - Programming im Endeffekt sehr viel effektiver, insbesondere auch bei der nachfolgenden Wartung der Releases. Zudem ist derjenige, der dem anderen zuschaut, erst einmal aus der Verantwortung, kann entspannt zuschauen, und sich dabei stressfrei auf den Code konzentrieren, der gerade vor seinen Augen vom Partner geschrieben wird. Es hat sich herausgestellt, daß Pair - Programming im Endeffekt die Kosten nicht erhöht, obwohl zu Beginn augenscheinlich die Leistung zweier Programmierer an einem PC geringer erscheint.

10. Unit Testing

Unit Testing ist bisher als selbstverständliche Praxis beim Programmieren noch nicht weit verbreitet. Ein Musterbeispiel ist Zope X3. Die neueste Zope - Version, siehe www.zope.org⁵⁰ ist vollständig nach den Kriterien des XP programmiert worden. Jeder Prozedur ist eine Test-Prozedur zugeordnet, die genau die Funktionalität überprüft. Allzu oft passiert es gerade beim Prozess des Refactoring, daß Funktionalität verändert wurde.

```
zope:~/ZopeX3-3.0.0/Dependencies# dtree
Initial directory = /home/zope/ZopeX3-3.0.0/Dependencies
+---BTrees-ZopeX3-3.0.0
|   +---BTrees
|   |   +---tests
+---Includes
+---RestrictedPython-ZopeX3-3.0.0
|   +---RestrictedPython
|   |   +---tests
+---ThreadedAsync-ZopeX3-3.0.0
|   +---ThreadedAsync
+---ZConfig-ZopeX3-3.0.0
|   +---ZConfig
|   |   +---components
|   |   |   +---basic
|   |   |   |   +---tests
|   |   |   +---logger
|   |   |   |   +---tests
|   |   +---scripts
|   |   +---tests
|   |   |   +---input
|   |   |   +---library
|   |   |   |   +---thing
|   |   |   |   |   +---extras
|   |   |   +---widget
|   |   .....
+---zope.app.cache-ZopeX3-3.0.0
|   +---zope.app.cache
|   |   +---browser
|   |   +---interfaces
|   |   +---tests
+---zope.app.dav-ZopeX3-3.0.0
|   +---zope.app.dav
|   |   +---ftests
|   |   +---tests
+---zope.app.debugskin-ZopeX3-3.0.0
|   +---zope.app.debugskin
+---zope.app.dtmlpage-ZopeX3-3.0.0
|   +---zope.app.dtmlpage
|   |   +---tests
+---zope.thread-ZopeX3-3.0.0
|   +---zope.thread
Total directories = 359
zope:~/ZopeX3-3.0.0/Dependencies#
```

Wie man sehen kann, sind Unit Tests inzwischen der Standard in der OpenSource - Bewegung. Immer mehr Teams setzen diese ein, weil die Softwarekomplexität immer höher wird, und immer mehr Code nach dem Refactoring eine veränderte Funktionalität aufweist, was nur UT entdecken können.

11. Graphische User Interfaces und Akzeptanz - Tests

Gute Bedienungsoberflächen programmieren, die vom User als "intuitiv" und "einfach zu bedienen" beschrieben werden, ist schwer. Jeder Programmierer hat hier zwangsläufig "Scheuklappen" auf - er weiß ja alles über das Programm und seine Bedienung - kann sich nur schwer in jemanden hinein versetzen, der keine Kenntnis über die Programmstruktur hat. Folgende Fragestellung hilft hierbei, die typischen Programmierer - Scheuklappen abzulegen, und sich in die Situation eines Users hinein zu versetzen:

- Eine Anwendung muß intuitiv zu bedienen sein, und in jeder Situation Hilfestellungen anbieten. Intuitiv bedeutet, daß die GUI das an Auswahlmöglichkeiten stets anbietet, was in der Erwartungshaltung des An-

wenders liegt. Eine GUI muß "erraten" können, was der Anwender denkt, bzw. sucht: Beim ersten Öffnen der GUI, was könnte der User an Informationen erwarten, welche benötigt er, was ist seine Absicht?

- Falls er in die GUI zurückkehrt (aus einem anderen Programmteil), welche Informationen benötigt er, um seine Arbeit fortzuführen, welche Informationen hat er z.B. vergessen? In früheren Programmiersprachen gab es ein "GOTO". Wie wäre es mit einem "COME-FROM"? Dies ist die Kunst der Gestaltung einer *statefull gui*.
- Die Gestaltung für Einsteiger und Fortgeschrittene muß unterschiedlich sein.
- Der Benutzer muß zu jedem Zeitpunkt erkennen können, in welchem Kontext er sich befindet.
- Er muß schnell und jederzeit in andere Kontexte wechseln, und wieder zurückkehren können (z.B. bei Unterbrechungen durch Telefonanrufe).
- Der Benutzer muß gelegentlich auf ungenutzte Features aufmerksam gemacht werden, wie z.B. schnelle Bedienung über Tastendrücke. Nutzt er diese, so müssen sich die Hilfestellungen verändern, reduzieren, oder abschalten.
- Die Anwendung muß sich dynamisch dem Benutzerverhalten anpassen. Häufig genutzte Funktionen müssen schneller erreichbar sein.
- Der Programmierer muß bei dem Design jeder Eingabe - und Ausgabemaske genau überlegen, was der Benutzer weiß, aus welchem Kontext dieser kommt, also welche Informationen dieser bereits hat, und welche er benötigen könnte.
- Teilausgefüllte Eingabe - Masken müssen stets wiederhergestellt werden, nach einem Kontextwechsel, oder sogar nach einem Reboot der WS.
- Eine GUI muß *statefull* sein, also genau wissen, welche Vorgänge vom User abgebrochen, welche nur unterbrochen, aber wieder aufgenommen wurden - ähnlich einer TODO Liste. Daten aus Eingabemasken müssen zwischengepuffert werden.
- Die Arbeit auf dem Desktop muß am nächsten Tag ununterbrochen fortgesetzt werden können, der User muß trotz Neustart seine Arbeit sofort fortsetzen können, siehe Terminal - Server.
- Eine Applikation in der GUI muß Multithreaded sein. Wartezeiten für z.B. Druckaufträge oder Reports dürfen nicht auftreten, die GUI muß den User über die erfolgreiche Ausführung eines Hintergrund - Taskes informieren.
- Eine GUI muß den User darüber informieren, wie fleißig dieser war (Datenerfassung, Hotline), woran er gearbeitet hat, welche Programme oder Teile der Anwendungsprogramme er nutzt oder nicht nutzt. Das Prinzip Wettbewerb (Agonalität) funktioniert auch hier.

12. Design Pattern

Was ist ein *Design - Pattern*? Betrachten wir ein Auto - wir haben Module - Motor, Türen, Elektronik, Stoßstangen, u.s.w. Wir haben Bezeichner dafür, und jeder weiß bescheid, was zu tun ist, wenn der Motor getauscht werden muß. Ohne Design - Patterns entwickelt, würde bei einem Auto, wenn man die Stoßstange abschraubt, die Kurbelwelle herausfallen, weil ein Ingenieur meinte, daß man das Kurbelwellenlager besser dort befestigen sollte, weil man dann im Falle eines Motorschadens nicht den Motor ausbauen muß, sondern nur die Stoßstange lösen muß.

- Martin Fowler definiert Design Pattern so: Ein Pattern ist eine Idee, die nützlich ist in einem praktischen Kontext (Zusammenhang), und vielleicht sich auch als nützlich in anderen erweisen kann.
- Gang Of Four definieren Patterns so: Design Patterns sind Beschreibungen zusammenarbeitender Objekte und Klassen, die massgeschneidert sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen.
- Christopher Alexander: Ein Pattern ist eine 3-wertiges Regelwerk, welche zwischen einem bestimmten Kontext, einem Problem und einer Lösung steht.

Wer Software neu entwickelt, tut sehr gut daran, sich im OpenSource - Bereich umzuschauen, viel Code zu lesen, und zu versuchen, die Pattern dahinter zu verstehen. Die Entwicklungszeit verkürzt nach erfolgreicher Übernahme der Pattern auf 1/3 oder 1/4. So viel schneller nämlich konnten z.B. die Programmierer von JBOSS, einem J2EE Clone die Software fertigstellen. Die J2EE - Programmierhandbücher von SUN waren die Grundlage. Ebenso wurde der Windows - Emulator Wine nur zu einem Bruchteil der Kosten nachprogrammiert. Zahlreiche Softwarepakete, darunter Apache - Module, wie PHPNuke, Postnuke, Zope, wurden mehrfach neu

geschrieben, jedoch in einer unglaublich kurzen Zeit, weil ja die wesentlichen Design - Merkmale schon fertig waren. Ebenso konnte ein freier .NET - Compiler MONO in einem kleinen Team in sehr kurzer Zeit nachprogrammiert werden. Philippe Kahn, der Gründer von Borland, hat in nur 6 Wochen Entwicklungszeit den TurboPascal - Compiler inklusive IDE, Debugger aus dem Boden gestampft. Dies funktioniert nur dann, wenn das Design schon vollständig im Kopf des Programmierers, zuende geplant, vorliegt. Je länger und intensiver die Planungsphase ist, je ausgereifter der erste Entwurf ist, umso schneller dann ist das erste Release fertig. Man beachte - in größeren Projekten, die viel innere Reibung haben, weil das Design erst noch entwickelt werden muß, trägt jeder Programmierer nur ganz 5 Zeilen Code je Tag bei. Jeder weiß aber, daß er durchaus 2-8 A4 Seiten Code je Tag schreiben kann, also 200-500 Codezeilen je Tag. Perfekte Planung vorausgesetzt, ist das Projekt dann Faktor 40-100 schneller fertiggestellt, so die Theorie. Die Übernahme von bereits vorhandenen Design - Pattern beschleunigt also die Software - Entwicklung bereits enorm. Unter diesem Aspekt sind z.B. folgende Patterns recht hilfreich:

Douglas C. Schmidt - Design - Patterns⁵¹, oder hier: http://designpatterns.ch/enterprise_patterns.html

Bildung ist das, was übrig bleibt, wenn man alle Details dessen, was man mal gelernt hat, wieder vergessen hat. Z.B. der Umgang mit Feuer - wer kann sich schon noch genau daran erinnern, wie und wann er den Umgang mit Feuer erlernt hat? Dennoch wissen wir scheinbar intuitiv, oder "unbewußt", wie man damit umzugehen hat. Bildung zeichnet sich dadurch aus, daß bewährte, antrainierte Denkmuster, Verfahrensmuster, Handlungsmuster bekannt sind, und angewendet werden. Z.B. wird ein Problem nach bewährtem Muster analysiert, um erkennen zu können, ob man alleine zur Lösung imstande ist, oder es ratsam sein könnte, weitere Experten zu befragen.

Denkmuster finden sich nicht nur in komplexen Handlungsabläufen oder Programmen, sondern man kann sie auch erkennen, z.B. in der Architektur von Häusern, Antroposophie, Philosophie (besonders Epistemologie), Soziologie, Gemeinschaften, Musik... Die ersten Anwender von Design Patterns auf Software waren von dem Forscher Christopher Alexander, Universität Berkeley, CA. beeinflusst, welcher Pattern allgemein erforscht hatte. Man kann es bei Kindern gut beobachten, die Häuser oder Autos malen - die teilweise abstrakten Linienzeichnungen sind eindeutig als Pattern erkennbar, man erkennt nämlich, was diese Linien bedeuten sollen. Mustererkennung ist uns Menschen angeboren, die Fähigkeit, diese niederzuschreiben, muß erlernt werden.

Software - Entwicklung ist teuer, jedoch steht nach umfangreichen Analysen von unzähligen Projekten (oft auch OpenSource) fest, daß weit über die Hälfte der Kosten nur dadurch entstehen, daß das Rad dauernd wieder neu erfunden wird. Bestehende Konzepte und Komponenten von Programmierern nicht genutzt, weil nicht bekannt, deren Wert nicht erkannt, oder einfach nicht verstanden, sodaß Konzepte schon während der Entwicklung teilweise wieder sich selbst überholt haben. So z.B. finden sich unter den Visual Basic (ACCESS) und PHP - Programmierern ein besonders hoher Anteil von Leuten, die einfach drauflos programmiert haben, relativ konzeptlos und unstrukturiert, mit dem Resultat, daß eine Applikation im Laufe der Entwicklung schon teilweise mehrfach umgeschrieben wurde. Diese Tatsache kann man sehr gut bei Access-Anwendungen und dem oft unverständlichen, kaum dokumentierten Spaghetticode beobachten, aber auch bei komplexeren PHP - Anwendungen der Open - Source - Gemeinde beobachten, wie z.B. PHPNuke, PostNuke, ... Hierbei sind mangels ausgereiften Konzepten riesige Code - Teile immer wieder umgeschrieben worden, von einem modularen Konzept, Wiederverwendung von Code ist hier nur wenig zu sehen. Erst in den neuesten Versionen erkennt man klare Design - Patterns und ein wohldurchdachtes Konzept. JAVA und insbesondere PERL Programmierer z.B. nutzen umfangreiche, komplexe Libraries (CPAN⁵³), die wie ein Zahnrad ins andere greifen, und dadurch bewährte Design Patterns schon oft vorgeben. Je mehr man auf ausgereifte Werkzeuge (Frameworks) zurückgreift, umso mehr wirken die bereits darin umgesetzten Design Patterns auf den neuen Code. Was bedeutet - je ausgereifter ein Entwicklungswerkzeug ist, desto mehr Design-Patterns finden darin Anwendung, und umso mehr können Programmierer darauf zurückgreifen. Ein Framework zeichnet sich besonders dadurch aus, daß eine hohe Zahl von bewährten, ausgereiften Design - Pattern enthalten sind, und mächtige Komponenten zur Verfügung stehen, sodaß die Programmierer diese nur noch "arrangieren" müssen, um zu lauffähigen Programmen zu kommen. Beispielsweise benötigt die Programmierung eines Web - Browser nur noch ca. 20-30 Zeilen Code. Dieser jedoch beherrscht dann eine Vielzahl von Protokollen, wie z.B. HTTP, FTP und vor allem WebDAV, sodaß dieser als Dateibrowser und Dokumentenverwaltungssystem direkt nutzbar ist. Hinter wohldurchdachten und leistungsfähigen Programmier - Frameworks stecken eine Unmenge von unentdeckten Features, die trotz der Einfachheit der Programmierung mit genutzt werden können, jedoch nicht müssen. Wer ahnt schon hinter der Benutzeroberfläche GNOME oder KDE ein Komponenten - Modell, oder wer ahnt, daß sich OpenOffice / Staroffice im Server - Modus starten lassen, sodaß man den Arbeitsplatz von einem Server aus (z.B. SAP/R3, Apache,...) aus fernsteuern kann ? Mehr hierzu in den entsprechenden Unter - Kapiteln.

Was ist die Essenz von Design - Patterns ?

- - repräsentieren konkrete Lösungsschemata für wiederkehrende Entwurfsprobleme
- - sind Konzepte oberhalb von Klassen und Objekten
- - basieren auf jahrzehntelanger Erfahrung und dokumentieren diese.

- - beschreiben Struktur und Verhalten interagierender Objekte
- - liefern ein gemeinsames Vokabular und Konzeptverständnis
- - bestimmen die qualitativen Eigenschaften von Problemlösungen

Ein Framework, in welchem viele dieser Design - Pattern Anwendung finden, zeichnet sich durch folgende, sichtbare Eigenschaften im Quellcode und der Art der Anwendungen aus:

- Trennung von Geschäftslogik, Workflow, Benutzeroberfläche
- Steuerung der Anwendung über andere Interfaces, außer der typischen Benutzeroberfläche für Anwender: Web, SOAP, XML-RPC, CORBA, PDA?
- Bedienungslogik für Web - Interface und Anwender GUI sind verschieden, dennoch bieten sie dieselbe Funktionalität.
- Geringe Abhängigkeit der Software - Module untereinander (Entkopplung), geringer Kopplungsgrad, klar definierte Schnittstellen.
- Steuerbarkeit des Workflow über externe Interfaces
- Möglichkeit zur Veränderung der Geschäftslogik bzw. Workflow über das Einbinden weiterer Module
- Veränderung und Steuerung der Grafischen Benutzeroberfläche im laufenden Betrieb möglich
- Unabhängigkeit vom Datenbankhersteller, Portabilität
- Klarheit, Wartbarkeit des Codes, Möglichkeit zum Refactoring mittels leistungsfähiger Werkzeuge, Versioning (CVS), Einbindung in die IDE
- Schnittstellen, Brücken zu anderen Systemen, z.B. TK-Anlage

13. Case Tools

CASE - Tools sind bewährte Werkzeuge, mit denen man schnell Software generieren kann, häufig für Spezial - Anwendungen. Haupteigenschaft von CASE - Tools ist es, daß die Bediener - und Anwendungslogik stark schematisiert wurde, also ein starres Design - Pattern hinterlegt wurde. Dieses starre Design - Pattern erst ermöglicht die schnelle Entwicklung von Software - allerdings mit dem Nachteil, daß die Kundenwünsche grundsätzlich nicht berücksichtigt werden, sondern sich der Kunde den Möglichkeiten des CASE - Tools anpassen muß. Soll das CASE - Tool erweitert werden, z.B. um mehr auf die Kundenwünsche einzugehen, so ist dies prinzipiell möglich, jedoch nur nach intensiver Einarbeitung in die internen Strukturen, die gewöhnlich bei normaler Anwendung verborgen bleiben. Die Erweiterung der vorhandenen, starren Design - Pattern birgt eine Gefahr: Ändert der Hersteller im neuen Release die darunterliegenden Strukturen, so ist ein Großteil der eigenen Arbeit vernichtet. Wann und wo man CASE - Tools einsetzen sollte, hängt von der Größe des Teams, den psychologischen Eigenschaften der Programmierer, dem Umfang der Aufgabe ab. Oft ist es sehr einfach, die Design - Pattern eines CASE - Tools selber nachzuprogrammieren bzw. danach zu erweitern. Sich in dem Markt der CASE - Tools umsehen lohnt sich.

14. Steven Reiss 16 Lebensmotive

Der Psychologe Dr. Steven Reiss deckte auf, dass so gut wie alles, was wir tun, auf 16 grundlegende Lebensmotive zurückgeführt werden kann. Mehrere dieser Motive bilden, mehr oder weniger stärker gewichtet, die Triebfeder im Leben. Steven Reiss Zusammenstellung ist über viele Jahre rein empirisch ermittelt worden.

Diese folgenden Aspekte entscheiden darüber, ob und wie ein Programmierer dauerhaft in ein Team eingebunden werden kann, und ob er in ein Team passt.

Die Auswertung mehrerer 100.000 Einzelaussagen, ohne konkrete, möglicherweise suggestiv wirkende Fragestellung, ergab folgende Aufstellung (ohne Wertung in der Reihenfolge):

- Macht: Streben nach Erfolg, Leistung, Führung und Einfluss
- Unabhängigkeit: Streben nach Freiheit, Selbstgenügsamkeit und Autarkie
- Neugier: Streben nach Wissen, Wahrheit, Erkenntnis
- Anerkennung: Streben nach sozialer Akzeptanz, nach Zugehörigkeit und positivem Selbstwert

- Ordnung: Streben nach Stabilität, Klarheit und guter Organisation
- Sparen: Streben nach Besitz und Anhäufung materieller Güter
- Ehre: Streben nach Loyalität und moralischer, charakterlicher Integrität
- Idealismus: Streben nach sozialer Gerechtigkeit und Fairness
- Beziehungen: Streben nach Freundschaft, Freude an dynamischen Prozessen und Humor
- Familie: Streben nach Familienleben und besonders danach, eigene Kinder zu erziehen
- Status: Streben nach Prestige, nach Reichtum, Titeln und öffentlicher Aufmerksamkeit
- Rache: Streben nach Konkurrenz, Kampf, Aggressivität und Vergeltung
- Eros: Streben nach einem erotischen Leben, Ästhetik, Sexualität und Schönheit
- Essen: Streben nach Nahrung
- Körperliche Aktivität: Streben nach Fitness und Bewegung
- Ruhe: Streben nach Entspannung und emotionaler Sicherheit

Der Begriff "Motivation", "Motiv" leitet sich aus dem lat. "motivare" ab, "von innen heraus bewegen". Neuere Forschungen der Biochemie haben ergeben, daß wir Menschen stets darauf bedacht sind, das Glücksgefühl im Leben (kurz- oder langfristiger) zu maximieren. Hierzu hat jeder Mensch sein eigenes "Belohnungssystem", welches, biochemisch betrachtet, den Pegel an Endorphinen, Dopaminen, ... im Blut hoch hält. Mehr hierzu siehe auch das *Buch von Josef Zehentbauer, Körpereigene Drogen*

Belohnungssysteme gibt es sehr viele:

- Gutes Essen, Trinken, Süßigkeiten
- Schmusen, Zärtlichkeit, Sexualität
- Flirten
- Kleine Geschenke als Anerkennungen
- Ein geschenktes Lächeln
- Spiele
- Pubertäres Schwärmen für Idole, Neugierde als Vorfreude auf Erkenntnis (Erkenntnis befriedigt)
- Musik/Film, Kunst, Sinnesfreude, Harmonie, Genuß der Ästhetik
- Tanzen, Party, Sport, langes Spaziergehen (setzt Endorphine frei)
- Prestige/Image, Luxus, Privilegien
- Träumen, Erinnerungen (Photos)
- Klatsch/Tratsch, Smalltalk, sich in Euphorie reden
- Inszenierung eines Streites (treibt Adrenalin hoch, danach erfolgt Endorphin/Dopaminausschüttung - dies ist mit ein Grund für zänkische Weiber, Nachbarschafts - Streit)
- Erfolgreiches Taktieren/Blöffen
- Lob/Anerkennung
- Alkohol/Drogen (insbesondere dann, wenn ein Mangel an körpereigenen Dopaminen vorliegt, kann in seltenen Fällen genetisch bedingt sein)
- Kreativ, schöpferisch sein (künstlerische Tätigkeit)
- Zum Frisör gehen (Frauen)
- Sich schick anziehen, ausgehen, präsentieren, repräsentieren, Neid erwecken
- Jemanden ärgern (Nachbarn, Arbeitskollegen, siehe Mobbing. Nach Adrenalinausschüttung erfolgt Endorphinausschüttung)
- Verleihen von Medallien, Pokale, Anerkennungen, Ehrenämter, u.s.w... in einem feierlichen Zeremoniell

Neben den direkten Belohnungssystemen gibt es noch indirekte, wie insbesondere die "Zugehörigkeit" zu einer bestimmten "Klasse", was verbunden mit entsprechenden Privilegien und Luxus, Prestige, Image eine Art

Dauerbelohnung darstellt. Der Stolz, zu einer bestimmten (privilegierten) Gruppe zu gehören, (Zugehörigkeit zur "upper class", siehe Tennis früher, heute Golf, aber auch Sippenzugehörigkeit) verschafft ein gutes Gefühl, ist also auch geeignet, den Pegel an Endorphinen zu steigern. Nicht umsonst verkaufen sich Käseblättchen wie warme Semmeln und deren Leser geben sich gerne dem Gefühl hin, "ihren Prommi" persönlich zu kennen, zur "upper class" zu gehören.

Im handwerklichen/beruflichen Bereich resultiert aus der Erziehung, daß Menschen besonders viel Befriedigung empfinden, wenn sie eine Arbeit gut, genau, pünktlich, oder sogar mit viel Sinn für die Details übererfüllt haben. In dieser Tradition in der Erziehung steht das deutsche Handwerk. Viele, kleine Details, die zeitaufwändig sind, wie z.B. das Entgraten von Bohrlöchern, damit sich niemand daran schneidet, machen den Ruf deutscher Handarbeit in der Welt aus: "Made in Germany". Ein deutscher Handwerker ist sich und seiner Arbeit etwas wert, empfindet Stolz, innere Befriedigung. Läßt man ein handwerkliches Produkt im Ausland herstellen, so ist es oft unmöglich, diesen Menschen den Sinn für Details zu vermitteln. Auch mit aufwändiger Qualitätsicherung und monetärem Belohnungssystem ist es vielen Firmen nicht gelungen, Produkte ähnlicher Qualität im Ausland zu produzieren.

Weniger bekannt ist der religiöse Hintergrund der handwerklichen Tradition: "Gott sieht alles!", "Vor Gott nicht mißliebig sein!" mahnt zu Qualität. Gekrönt wird die Handwerksausbildung mit der rituellen Aufnahme in die Innung, die auch heute noch mit einem Kirchgang und großzügiger Spende an die Kirche verbunden ist. Beruf kommt von "Berufung", von "Gott berufen". Es ist eine Geisteshaltung, eine innere Einstellung, ein Belohnungssystem, welches im Rahmen der religiösen Tradition anerzogen wurde.

Erziehung der Kinder z.B. zum Neid auf materielle Güter ist eine von vielen Eltern angewandte Methode, welche den späteren Ehrgeiz wecken soll, einen guten Beruf zu erlernen, sich was leisten und gönnen zu können. Betrachtet man die 16 Grundmotivationen von Steven Reiss in diesem Zusammenhang ist klar, welche Art der Erziehung man selber genossen hat, und welche Folgen dies auf die Berufswahl und die Art der Berufsausübung (angestellt, selbstständig) hatte.

In Firmen gibt es verschiedenste Belohnungssysteme, sie machen einen Teil der Unternehmenskultur aus. Natürlich reagiert nicht jeder Mitarbeiter auf dieselben Belohnungssysteme. Mancher drängt sich gerne in den Vordergrund, sucht seine Belohnung in der öffentlichen Anerkennung seiner Leistungen, ein anderer Mitarbeiter ist eher ein Blümchen, welches im Verborgenen blüht, qualitativ hochwertige Arbeit leistet, ohne viel Aufhebens darum zu machen. Dienstwagen, Sachleistungen, Sonderzulagen, leistungsabhängige Entlohnung sind Anreizsysteme der monetären Art, auf die nicht jeder Mitarbeiter gleich reagiert. Die schon in der Erziehung beginnende Vermittlung des Selbstwertgefühls entscheidet über die Identitätsaufwertung/erhöhung, die jemand für sein persönliches Glückempfinden benötigt.

15. Lösung des Zwergenproblems

Die Zwerge stellen sich auf der Linie auf. Zunächst nur zweie. Angenommen, diese hätten dieselbe Mützenfarbe, rot: RR. Nun stellt sich ein Zwerg daneben, mit grüner Mütze: RRG. Alle weiteren Zwerge wählen nun, weil sie selber ja nicht genau wissen, welche Farbe ihre Mütze hat, genau die Grenze zwischen Rot und Grün, wobei die anderen auseinanderrücken. Während zunächst jeder Zwerg Betrachter ist, wird er zum Betrachteten. Die implizite Logik dabei ist also, daß er seine Rolle wechselt, in dem Moment, wo er sich auf der Linie genau auf der Grenze zwischen Rot und Grün aufstellt. Es besteht daher kein Bedarf, daß jemand die Zwerge aufteilt, Kommunikation untereinander ist ebenfalls nicht nötig, um zum Ziel zu gelangen, aber nur dann, wenn wirklich jeder das Prinzip verstanden hat, genau weiß, was er zu tun hat, und vor allem - mitdenkt und diszipliniert handelt:

```
RR
RRG
RRGG
RRGGG
RRRGGG
RRRRGGG
RRRRGGGG
...
RRRRRRRRGGGGGGGGGG
```

Dieses Beispiel soll anschaulich machen, daß sehr viel unnötiger Kommunikationsaufwand bei der Programmierung (Abstimmungen, Teambesprechungen, Kaffeekränzchen mit Kuchen) vermieden werden kann, wenn allgemein mitgedacht wird. Jeder denkt für den anderen mit, macht ihn frühzeitig auf Fehler aufmerksam, verhindert so aufwändige Korrektur - und Nachbesserungsarbeiten. Und nun noch ein weiteres Rätsel: "Wie sieht die Zwergenlösung aus, wenn 3 Hutfarben im Spiel sind?". "Was folgt daraus für die Lösung mit 2 Hutfarben?".

Fußnoten

1. <http://www.little-idiot.de/his/his.pdf>
2. <http://www.little-idiot.de/philosophie/AxiomenSysteme.pdf>
3. <http://www.little-idiot.de/philosophie/AutreCourt.pdf>
4. <http://www.freshmeat.net>
5. <http://www.sf.net>
6. <http://www.fedora.org>
7. <http://www.mozilla.org>
8. <http://www.openoffice.org>
9. <http://www.compiere.org>
10. <http://www.postgresql.org>
11. http://www.janus-software.com/fyracle_demo.html
12. <http://www.phpnuke.org>
13. <http://postnuke.org>
14. <http://www.zope.org>
15. <http://www.opencms.org>
16. <http://www.little-idiot.de/linuxsolutionguide/softwareentwicklung.html>
17. <http://www.netscape.org/>
18. <http://www.ca.com/>
19. <http://www.mysql.com/>
20. <http://www.little-idiot.de/teambuilding/ImpliziteLogikenTeamgeist.pdf>
21. <http://www.phpbb.de>
22. <http://www.otrs.org>
23. irc.freenode.net
24. <http://www.doxygen.org>
25. <http://unstable.elemental.com/mozilla/build/latest/mozilla/dom/dox/interfacensIDOMHTMLPreElement.html>
26. <http://www.eclipse.org>
27. <http://bugzilla.samba.org>
28. <http://www.mailing-archive.com>
29. <http://www.google.de/>
30. <http://groups.google.de/>
31. <http://www.bugzilla.org>
32. <http://www.egroupware.org>
33. <http://www.phpnuke.org>
34. <http://www.postnuke.org/>
35. <http://www.freshmeat.net/>
36. <http://www.sf.net>
37. <http://www.cvshome.org/>
38. <https://www.cvshome.org/dev/addons.html>
39. <http://www.bitkeeper.com>
40. <http://www.little-idiot.de/linuxsolutionguide/book1.htm>
41. <http://www.little-idiot.de/linuxsolutionguide/>
42. pythius.sf.net
43. <http://www.cs.tut.fi/~ask/aspects/aspects.html>

44. <http://www.logilab.org//projects/aspects/>
45. <http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/starting.html>
46. <http://www.cs.ubc.ca/labs/spl/papers/>
47. <http://www.extremeprogramming.org>
48. <http://www.little-idiot.de/teambuilding/kaizen.pdf/>
49. <http://www.little-idiot.de/teambuilding/VonChinaLernen.pdf>
50. <http://www.zope.org>
51. <http://www.cs.wustl.edu/~schmidt/patterns.html>
52. http://designpatterns.ch/enterprise_patterns.html
53. <http://www.cpan.org>