Soziale Strukturen in Software Projekten

Dr. Bernhard Scheffold
Walter Kriha

Soziale Strukturen in Software Projekten

by Dr. Bernhard Scheffold and Walter Kriha

Published 19.Feb.1998

Table of Contents

Chapter 1. Abstract

Der Vortrag versucht – basierend auf eigenen Erfahrungen sowie dem Austausch mit Kolleginnen und Kollegen – soziale Strukturen in neuen Softwareprojekten zu beleuchten. Dazu gehört ein Blick auf das „Alte", gegen das sich das „Neue" häufig technisch und sozial abgrenzt. Anhand von zunächst blossen Auffälligkeiten (z.B. Identität und Produkt, neue Leute – neuer Ort) wollen wir zeigen, dass es die sozialen Strukturen sind, die gewisse Muster in Softwareprojekten erzeugen, teilweise sogar zur Replizierung von Zuständen führen, gegen die das „Neue" ursprünglich angetreten war.

Da wir selbst in der Softwareentwicklung tätig sind, geht unser Interesse natürlich über die blosse Feststellung von sozialen Strukturen hinaus. Im zweiten Teil untersuchen wir deshalb den Zusammenhang von sozialen und technischen Strukturen. Woran scheitert z.B. das Wasserfallmodell der Projektentwicklung wirklich? Warum liegen die grössten Softwareprobleme gerade in den Bereichen Zuverlässigkeit, Erweiterbarkeit und Qualität? Sind technische Interfaces die richtige Stelle zur sozialen Giederung? Wie beinflussen die technisch-sozialen Strukturen unsere Kategorisierungsweisen? Wie sind die Kriterien bei der Auswahl von Mitarbeitern für neue Projekte?

In diesem Zusammenhang betrachten wir den Einfluss hochgradiger Arbeitsteilung auf die technischen Strukturen eines Projektes oder Produktes über seinen Lebenszyklus hinweg.

Abschliessend wollen wir einen Blick auf zukünftige Formen der Softwareentwicklung – z.B. das Entwickeln per Internet oder das extern entwickelte Mega-Projekt) werfen, um zu sehen, ob die vorher entwickelten Erklärungsmuster zumindest teilweise auch hier verwendbar sind. Oder sind es gar die ungelösten sozialen Probleme herkömmlicher Softwareprojekte, die zu massivem Outsourcing bzw. dem Entwickler als selbständigem Unternehmer führen?

Chapter 2. MANAGEMENT-SUMMARY

Neue Softwareprojekte werden häufig allein unter dem Aspekt der Einführung neuer Technologie gesehen. Am – häufig erfolglosen – Ende solcher Projekte stellt man aber fest, dass es weitere Kriterien geben muss, die das Projekt bzw. seinen Erfolg massgeblich beeinflusst haben. Ein Vergleich persönlicher Erfahrungen lässt schnell erkennen, dass es soziale Dinge sind, die das Projekt im wesentlichen ausmachen. Und sie sind keineswegs amorph und undurchschaubar, sondern tauchen meist in Form von Mustern auf.

Im folgenden versuchen wir den sozialen Hintergrund von Projekten zu beleuchten und Vorschläge zur Verbesserung zu machen.

Zuerst untersuchen wir „typische" Abläufe, d.h. immer wiederkehrende Verfahrensweisen. Sie sind zunächst nichts weiter als Auffälligkeiten. Im zweiten Schritt versuchen wir Erklärungen zu einzelnen dieser Auffälligkeiten sowie einzelne Verbesserungsvorschläge zu liefern. Der dritte Schritt versucht eine strukturelle Sichtweise der sozialen Fundierung von Softwareprojekten. Damit wollen wir erklären, wieso das neue Projekt nach einiger Zeit genau die gleichen Probleme aufwirft wie dasjenige, das es abgelöst hat. Wieso sich der Kreislauf wieder schliesst.

Chapter 3. WIESO MIT SOZIALEN STRUKTUREN BESCHäFTIGEN?

Dafür gibt es in unserem Fall mehrere Gründe. Hauptgrund waren sicherlich unsere Erfahrungen im letzten Projekt, einer Framework-Entwicklung die ein bestehendes Produkt ablösen sollte. Das – letztlich gescheiterte – Projekt hat uns noch lange Zeit, nachdem wir bereits die Firma verlassen hatten, beschäftigt. Noch Monate danach auf dem Weg zur Arbeit der tägliche innere Dialog mit ehemaligen Kollegen – warum haben sie dies oder jenes einfach nicht erkennen wollen? Woran ist das Projekt letztlich gescheitert? Etc.

Wir haben dann angefangen – vielleicht als nachträgliche Rechtfertigung – die technischen Merkmale einer Framework-Entwicklung niederzuschreiben. Was sollte besser werden? Wie sollte es gemacht werden? Welche Programmiertechniken waren nötig? Welche Abstraktionen und Modelle mussten entstehen?

Im Laufe dieser – sehr technischen Auseinandersetzung – wurde jedoch immer deutlicher, dass ein guter Teil des Scheiterns gar nicht technisch bedingt gewesen war. Wie hätte es uns auch sonst noch Monate danach beschäftigen können?

Fragen an gescheiterte Projekte

- Was hat die "Neuen" von den "Alten" unterschieden?
- Welche Modellbildungen waren unannehmbar?
- War der neue Arbeitsstil so anders (basierend auf Kommunikation und gemeinsames Verständnis)?
- Wieso fällt es so schwer, von alten Konzepten Abschied zu nehmen?
- Wie ist der Zusammenhang zwischen technischen Strukturen und Konzepten und sozialen Strukturen?
- Welche Rolle spielt die Arbeitsteilung in der Software-Entwicklung?
- Sind Grundprobleme der Software-Entwicklung wie Zuverlässigkeit, Erweiterbarkeit, Wartbarkeit und Qualität letztlich soziale Phänomene?
- Wieso scheinen sich bestimmte Probleme in Softwareprojekten immer zu wiederholen?

Ergebnis: Strukturen und Verbindungen in einem Framework – Ein Bekenntnis zu Abstraktion und Komplexität entlang der Zeitachse von Projekten.

Neben diesem eher persönlich motivierten Grund gibt es weitere:

Anerkennung sozialer Faktoren wird "legal"

- Design-Pattern-Bewegung
- Soziale Tauglichkeit von Programmiersprachen
- Firmenorganisation als Framework
- Weitere Sichtweise von Software-Architektur: Neue Rollen, Interaktionsmuster, Denkmuster und Kultur der Entwicklung von Software

Es ist jetzt legal für Software-Entwickler, sich mit den sozialen Bedingungen ihrer Arbeit zu beschäftigen. Die Design-Pattern-Bewegung stellt z.B. klar fest, dass 50 % des Wertes von Design-Patterns sozial sind: Sie ermöglichen es, die eigene Arbeit zu kommunizieren. Sie ermöglichen es darüber hinaus, fremde Arbeit aufzunehmen und in der eigenen Arbeit zu verwenden. Firmen wie die Bell Labs schicken Spezialisten für objektorientierte Technologie (z.B. Jim Coplien) durch ihre Abteilungen, um Design-Patterns aufzuspüren und die sozialen Bedingungen ihres Einsatzes zu untersuchen.

Sogar die soziale Tauglichkeit von Programmiersprachen wird z.B. von Richard Gabriel in Frage gestellt: Kann mit ihnen Wissen kommuniziert werden? Oder dienen sie mehr dem Ego ihrer Erfinder?

Die Verbindung von Technik und Sozialem geht aber auch anders herum: Firmen wie die Systor AG bezeichnen ihre Organisation in bewusster Analogie als Framework. Hier werden Konzepte der Softwareentwicklung auch für die interne soziale Organisation der Firma angewendet.

Nicht zuletzt Mut gemacht haben uns auch die Erfahrungen mit dem IIG. Dr. Strubes Konzept der „Situation" im Design von graphischen User-Interfaces hat uns ein wesentlich besseres Verständnis ermöglicht (uns die nötige Sprache gegeben). Und die Ergebnisse der Studien zum Frauenanteil an Informatikstudenten waren auch für Nicht-Fachleute nachvollziehbar; vielleicht gerade weil sie im Herangehen sowohl verstehend als auch empirisch sind.

Die Anerkennung der Softwareentwicklung als sozialer Tätigkeit ist dennoch noch längst nicht überall erfolgt.

Chapter 4. DAS LEIDEN AM ALTEN UND DER WEG ZUM NEUEN

My experience is that management distrusts developers and feels held hostage by them. I have heard many discussions in which software-company CEOs look forward to the day when software development does not require hiring those pesky programmers.[...]

You know, sometimes it‘s a people problem, not a technology problem.

(Richard Gabriel, Patterns of Software, pg. 131)

Neue Softwareprojekte setzen alte logisch voraus. Existierende Projekte bestimmen und beeinflussen neue Projekte jedoch in viel subtilerer Weise. Oft werden die Ziele des neuen Projektes sowie die verwendete Technologie gerade durch die Erfahrungen mit den existierenden Systemen und deren Entwicklern bestimmt. Die Kritik am „Alten" erfolgt jedoch in mehreren Dimensionen und auf unterschiedliche Weise. Business, Technik, Soziales und Denkmuster werden unterschiedlich offen und mit unterschiedlichem Grad an Detail und Wissen diskutiert und kritisiert.

Dimensionen der Kritik am Alten

Formen der Kritik am Alten

Business	Technik	Sozial	Denkmuster
Offen	Offen	Implizit oder gar nicht	kaum
Im Detail	Im Detail, aber oft nicht über den Lebens-zyklus	Kaum Ursachen-forschung	Kein Wissen, Leiden

- Explizite Kritik am System, meist vorgetragen in Form von abgelehnten Wünschen der Benutzer
- Implizite Kritik an der Organisationsform der Entwicklung und den starren Konzepten und Weltbildern der Beteiligten. Obwohl diese Kritik selten offen geäussert wird, spielt sie eine grosse Rolle.

Woher wissen wir über die implizite Kritik am Alten? Ohne diese sind bestimmte Phänomene neuer Projekte (s.u. Beobachtungen, Neues Projekt) einfach nicht zu verstehen.

Was wird explizit bemängelt?

Explizite, offene Kritik am Alten

- Kosten
- Mangelnde Flexibilität
- Fehler, Qualität
- Schwierige Handhabung
- Aufwendige Installation und Wartung
- Nicht mehr zeitgemäss
- Mangelnde Kapselung macht Änderungen schwierig und gefährlich
- Keine Reuse der Software

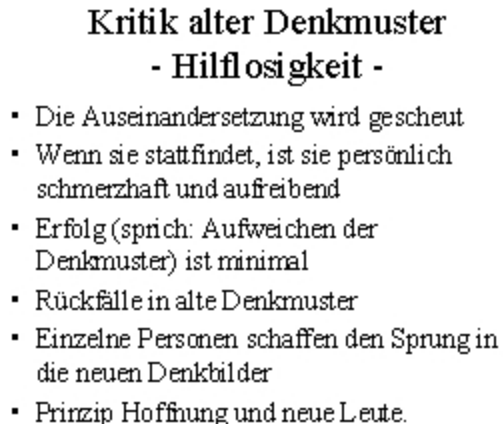
In den seltensten Fällen erfolgt jedoch eine genaue Analyse der Stellen im System, die diese Probleme verursachen. Damit bleiben grundsätzliche, strukturelle Probleme verborgen und den Verfechtern des alten Systems bleibt immer der Ausweg, dass man dieses oder jenes Merkmal ja noch nachrüsten könne. Diese Auslassung kann fatal sein: Lässt sie doch das alte Konzept bzw. System als mögliches Paradigma weiter bestehen und stellt gleichzeitig die Notwendigkeit des neuen Systems bzw. des neuen Denkens in Frage.

Bei genauerem Hinsehen (sprich eigener Erfahrung mit dem alten System) zeigen sich jedoch schnell auch organisatorische und soziale Kritikpunkte, die jedoch im allgemeinen höchstens hinter vorgehaltener Hand ausgesprochen werden.

Implizite, versteckte Kritik am Alten

- Verwalter des alten Systems sind arrogant
- Sie wollen keine Veränderungen
- Keine benutzerfreundliche Organisation
- Gängelung statt Unterstützung
- Undurchschaubare, zirkuläre Argumente
- Technik dominiert Business

Der letzte Kritikpunkt richtet sich meist auf die starren Konzepte und Denkkategorien der bisherigen Entwickler, die öfters in direktem Widerspruch zu Business-Strategien geraten.



Genau wie die sozialen und organisatorischen Kritikpunkte werden auch die alten Denkmuster kaum offen angegriffen. Im Umgang mit ihnen offenbart sich die grösste Hilflosigkeit. Meist bleiben im Umgang mit alten und neuen Denkmustern nur Wege der Trennung.

Im folgenden wollen wir zeigen, welche soziale Strukturen ein neues Softwareprojekt entwickelt, bzw. von welchen es getrieben wird. Unsere Behauptung ist, dass diese Strukturen letztlich dazu führen, dass aus dem Neuen wieder ein Altes wird, das wieder nur durch eine Revolution abgelöst werden kann. Die Replikation der sozialen Strukturen als Ursache dafür, dass es in der Softwareentwicklung scheinbar immer nur Revolutionen und kaum Evolution gibt.

Die Allgemeinheit des Leidens

Nachdem wir diese Erfahrungen aus einer relativ kleinen Firma niedergeschrieben und an Kollegen in Grossprojekten und Grossfirmen verteilt hatten, war die für uns erstaunlichste Rückmeldung, dass dies genau die Vorgänge bei Grossprojekten mit Milliarden DM Budget widerspiegeln würde.

Ist das Leiden am Alten und der Weg zum Neuen wirklich unabhängig von der Grösse der Firmen?

Chapter 5. BEOBACHTUNGEN

Hier wollen wir einfach technische und soziale Dinge, die uns aufgefallen sind, aufführen. Dies geschieht unter der Annahme, dass sich soziale Strukturen in der Softwareentwicklung nicht nur in Organigrammen niederschlagen, sondern quasi auf einer Micro-Ebene das alltägliche Handeln der Softwerker beeinflussen. Anschliessend wollen wir die Bobachtungen zu Mustern zusammenzufassen und Erklärungen bieten.

Neues Projekt: Neue Leute

Oft stellt eine Firma für ein Softwareprojekt mit neuer Technologie neue Leute ein. Wieso soll/kann es nicht mit den alten Mitarbeitern angegangen werden? Vermutung: Es hat mit einer „hidden agenda", der oben angesprochenen Unzufriedenheit nicht nur mit technischer Leistung der alten Produkte, sondern auch mit der sozialen Organisation der Leistung und alten Denkbildern, zu tun.

Neue Leute bedeuten eine – noch – offene soziale Struktur. Es lässt sich noch etwas bewegen, da die Rollen noch nicht fest verteilt sind.

Neue Leute werden so ausgesucht, dass sie die „neuen" Denkmuster und Konzepte bereits mitbringen. Ein stilles Eingestehen der Tatsache, dass die alten Mitarbeiter dazu nicht mehr zu bewegen sind.

Immer häufiger werden neue Projekte jedoch auch durch „outsourcing" gestaltet, bzw. die neuen Leute werden nicht als Mitarbeiter angestellt, sondern stehen in einem Selbständigen-Verhältnis. Wieso entscheiden sich Firmen dazu, ehemalige Kerngebiete ganz in die Verantwortung von Firmenfremden zu geben? Unsere Vermutung ist, dass dahinter ein stilles Eingeständnis des Managements liegt, dass sich das neue Projekt im eigenen Haus wieder zum gleichen Problemfall wie das alte entwickeln würde. Teilweise stehen wohl auch grundsätzliche Erkenntnisse über den Konflikt zwischen eigenem Business und Eigenentwicklung von Software dahinter (siehe Architekturen und soziale Gliederung weiter unten).

Neues Projekt: Ortswechsel

Symbolik des Raumes – ein in der Literatur oft genutztes Mittel. Tatsache ist, dass viele Leiter neuester Technologieprojekte sich dazu entscheiden, aus dem bestehenden Standort wegzugehen. Der Verlust bestehender Infrastruktur wird ausgeglichen durch mehr Flexibilität und dem leichteren Aufbau einer neuen Gruppenkultur in Abgrenzung zum Bestehenden. Neuer Raum vermittelt das Gefühl von mehr Dynamik. Sonderrechte sind leichter zu geben, was unter den Mitarbeitern zur Mehrarbeit führt – oft viel mehr als unter den Mitarbeitern die unter dem standardisierten und kontrollierten Regime des Hauptsitzes arbeiten.

Die Alten und die Neuen: Krieg ohne Genfer Konvention

Eine soziale Struktur die schnell sichtbar wird, genauso wie das Spannungsfeld, das sie ergibt.

Was zeichnet die alten Mitarbeiter aus?

Das Wissen um ein bestehendes Produkt, oft in Form eines geheimnisumwitterten Spezialwissens um obskure Details der Implementation. (Wieso ist dieses Wissen etwas besonderes? Liegt es nicht offen zu Tage? Ist EDV immer noch hauptsächlich blosses Wissen von Sachverhalten?)

Erfolge – das alte Produkt existiert und wird verkauft.

Eine bestehende Gruppe, die sich seit längerem kennt, eine Gruppenkultur entwickelt hat. Offene Frage: Wieso braucht es die Neuen? Wieso haben die Alten das neue Projekt nicht bekommen? Ist dies eine indirekte Kritik der Geschäftsleitung an ihnen? Ist das alte Produkt, das so erfolgreich war, plötzlich nicht mehr gut genug? Man könnte doch das alte genauso weiterentwickeln! (Siehe Thomas Kuhn zur Struktur wissenschaftlicher Revolutionen).

Sollen die Neuen mal sehen, wie sie mit den Problemen der Anwendungsdomäne klarkommen.

Fragen: Warum hat man die neuen Kenntnisse nicht erworben? Weil man zu beschäftigt war, die Firma aufzubauen, das momentane Produkt zu entwickeln. Jetzt wo man es geschafft hat, kommen neue Leute und ernten die Früchte.

Was zeichnet die Neuen aus?

Fehlendes Detailwissen des bestehenden Systems. Oft neuere technische Kenntnisse. Oft viel geringere Kenntnisse der Applikation-Domain.

Ergebnis: Verweigerung der Zusammenarbeit. Information-Hiding (im Sinne der Vorenthaltung von Informationen). Heimliches Weiterentwickeln des alten Systems. Grabenkämpfe, Koalitionen. Die neuen Leute stehen vor zwei Alternativen: Versuch der Kooperation oder Alleingang unter Duplizierung vorhandenen Wissens. Wenn gleichzeitig ein Ortswechsel vollzogen wurde, hat die Leitung des neuen Projektes entschieden, den Problemen aus dem Weg zu gehen.

Eine neue Sprache und ein neues Weltbild entstehen.

Hauptmerkmal neuer Entwicklungen ist fast immer eine neue Sprache, ein neues Kategorisierungssystem.

Aus einem Architekturdokument einer Grossbank:

Die Informationsstruktur [..] versteht sich als Kommunikationshilfsmittel und vermeidet daher möglichst Begriffe, die heute schon zu Missverständnissnen führen. Insbesondere werden bewusst neue Begriffe gewählt, wenn durch die Weiterverwendung eines etablierten Begriffs der Umfang eines Themas eingeschränkt würde.

Die Informationsstruktur bildet letztich das Gliederungskriterium für die Bildung von Service-orientierten Kompetenzzentren.

An dieser Aussage sind mehrere Dinge bedeutsam:

1. Die Erkenntnis, dass Begriffe eine kommunikative Bedeutung haben und nicht nur zur Reduktion auf ein objektivistisches Konzept von Wahrheit dienen.

2. Der Wunsch nach Erweiterung der Denkbilder und Kategorien (z.B. Dokument statt Beleg).

3. Die sofortige soziale/organisatorische Aufteilung der Mitarbeiter entlang der begrifflichen Grenzen. Die begrifflich-technische Modellbildung steuert die soziale Gliederung. (Dazu mehr weiter unten unter dem Stichpunkt „Technische Modelle und soziale Gliederung")

Der Kampf um die Worte nimmt manchmal fast lächerliche Dimensionen an, wenn z.B. in einem Meeting die Alten beharrlich von „Beleg" sprechen, die Neuen jedoch von „Dokument". Es wäre zu überlegen, ob eine Sprachanalyse den Grad der Integration neuer Technologie in einer Firma anzeigen könnte. Wenn z.B. nach längerer Entwicklung die alten Mitarbeiter immer noch von „Beleg" sprechen, dann darf die

Integration der neuen Technologie angezweifelt werden, und es liegt ein enormes Problem für die Firma vor.

Wieso entsteht die neue Sprache? Eine Technologie bringt meist eine neue Sichtweise des Gegenstandsbereichs mit sich. Diese Sichtweise kann die Begriffe des alten Systems nicht verwenden, da diese eben an das alte System gebunden sind. Es würden Doppeldeutigkeiten entstehen und die Möglichkeit von neuer Abstraktion und Verallgemeinerung genommen.

Das Ausmass der unterschiedlichen Sprachwelten lässt sich z.B. daran erkennen, dass die Architekturdokumente zur Softwarearchitektur von SBC und UBS sich jeweils pro Technologie gleichen, d.h. die Unterschiede zwischen den Firmen sind weitaus geringer als zwischen den Technologien.

Was uns teamfähig macht: Autisten und Teamplayer

Selbstbilder:

Das Bild vom (meist männlichen) Einzelkämpfer, der im stillen Kämmerlein grossartige Software entwickelt. Träume von Macht und Kontrolle. Ungeduld ist das zentrale Kennzeichen vieler in der EDV Tätigen – alles ist so mühsam und dauert ohnehin so lange. Kommunikation und Wissenweitergabe sind mühsam, halten von der Arbeit ab. Abstimmen und auf andere angewiesen zu sein ist unangenehm.

Gruppenkultur: Schweigeorden und die Dominanz einzelner.

Immer wieder stösst man auf das Phänomen der Gruppenkultur in Entwicklungsprojekten. Einzelne Personen dominieren so stark, dass sie ein bestimmtes Gruppenverhalten erzwingen. Im Extremfall entsteht ein Schweigeorden.

Fragen:

Wieso kommt die Dokumentation immer am Ende?

Welchen Einfluss hat die Ausbildung (Mathematiker, Physiker, Ingenieure)?

Pioniere als Generalisten oder warum es wichtig ist, der Erste zu sein

Bei den Ersten eines neuen Softwareprojektes zu sein, ist ein grosser Vorteil: Arbeitsteilung ist noch nicht so ausgeprägt. Man ist gezwungen, in vielen verschiedenen Gebieten zu arbeiten, Grundlagen zu schaffen, Überblick zu verschaffen. Die Organisationsstrukturen sind unscharf, offen, flexibel.

Man erwirbt schlichtes Faktwissen (die DLL xyz.dll arbeitet intern mit 2-byte-alignment, weil die verwendete externe Library es ursprünglich mal so benötigte).

Später Eingestellte treffen in mehrfacher Hinsicht auf starrere Strukturen: Die Software ist bereits entstanden, vorstrukturiert. Und die Organisationsstruktur hat nur noch spezifische Leerstellen („Wir brauchen noch eine zweite Person für die Datenbank") mit genau definierter Funktionalität. Und es gilt das „Closed shop"-Prinzip: Projekte schweissen zusammen, es entsteht eine Gruppenkultur die es dem Neuen u.U. extrem schwer macht einzusteigen. Verhaltensregeln, oft unausgesprochen, sind in Kraft.

Ergebnis: Die Trennung in Alte (altes Projekt) und Neue (neues Projekt) fängt nach kurzer Zeit an, sich auch im neuen Projekt zu replizieren.

Verhaltensmuster und Typen: Pragmatiker, Praktiker und Spinner.

Der Erfolg eines Projektes hängt entscheidend von der Auswahl der Mitarbeiter/Beteiligten ab. Bestimmte „Typen" von Entwicklern sind recht häufig anzutreffen und müssen daher bereits am Anfang eines Projektes identifiziert werden.

Wieso gelingt es in der Softwareentwicklung Leuten, sich als „Reichsbedenkenträger" durchs Leben zu schlagen? Es gelingt ihnen, sich ohne jede Abstraktionsfähigkeit durch komplexe Projekte zu schlängeln indem sie:

1. sich als militante Praktiker/Pragmatiker bezeichnen und jede Frage nach Konsequenzen bestimmter Techniken als „theoretisches Gefasel" abtun.

2. Zu jedem technischen Vorschlag „grösste Bedenken" äussern. Typischerweise wird ganz generell „die Performance" als kritisch angegeben. Dies ergibt – zumindest für einige Zeit – den Nimbus von Erfahrung und Professionalität. Bevor die Konsequenzen von Projekten in all ihren schaurigen Details sichtbar werden: in ein anderes Projekt wechseln.

Gibt es in der Softwareentwicklung kein prüfbares Wissen? Wie schützt man sich vor solchen „Pragmatikern"? Z.B. indem man in Bewerbungsgesprächen auf Wissen über Design-Patterns abprüft und versucht den „Performance"-Bewussten zu identifizieren.

Ein weiterer häufig anzutreffender Typ ist der „Praktiker", der der Performance alle anderen Kriterien der Softwareentwicklung (wie z.B. Wartbarkeit und Erweiterbarkeit) unterordnet. Kennzeichnend für diesen Typ ist, dass er die Entwicklung der Hardware-Technologie hin zu weit schnelleren Systemen keinesfalls als Grund ansieht, seine Denkweise anzupassen.

Der Augen-zu-und-durch-Typ: Kennzeichnend für ihn ist dass er grundsätzlich keine Lebenszyklus- oder Systembetrachtung anstellt. Identifizieren lässt er sich dadurch, dass er sich als „Minimalist" bezeichnet, d.h. es genügt die Kenntniss einer Programmiersprache um Softwareentwickler zu sein. Man kann alles damit machen.

Risiko und Technologie – soziale Voraussetzungen des „inventors paradox"?

In my life as an architect, I find that the single thing which inhibits young professionals, new students most severely, *is the acceptance of standards that are too low*. If I ask a student whether her design is as good as Chartres, she often smiles tolerantly at me as if to say, `Of course not, that istn‘t what I am trying to do... I could never do that.‘

Then, I express my disagreement and tell her:`That standard *must* be our standard. If you are going to be a builder, no other standard is worthwhile. That is what I expect of myself in my own buildings, and is what I expect of my students.‘ [..] Once that level of standard is their minds, they will be able to figure out, for themselves, how to do better, how to make something that is as profound as that. (Christopher Alexander, Vorword to Patterns of Software by R.G.Gabriel)

Die Annahme, dass eine bestimmte Technologie entweder richtig oder falsch ist, ist für die Praxis der Softwareentwicklung ziemlich bedeutungslos. Entscheidend ist, ob eine Technologie eingesetzt wird. Und dies wiederum ist sozial determiniert.

Beispiel:

Jeder Entwickler merkt ziemlich schnell, dass er oder sie immer wieder vor derselben grundlegenden Entscheidung steht:

Es gibt ein Problem. Für dieses Problem gibt es eine Menge von Lösungen mit jeweils unterschiedlichen Kosten, Benefits und Entwicklungszeiten. Das „inventors paradox" behauptet nun, dass die allgemeingültige Lösung sich als die einfachere Lösung herausstellen wird. Dies beruht auf dem Effekt der Abstraktion, der Gemeinsamkeiten von Problemen und Lösungen sichtbar macht..

Anscheinend sollte es möglich sein, durch Schätzung den Aufwand zu ermitteln, die eigenen Requirements aufzustellen und danach zu einer rationalen Entscheidung zu kommen. Eine Folge davon müsste sein, dass recht häufig die generische Lösung genommen wird, weil sie eben einfacher und flexibler ist als Investition.

Tatsächlich fällt die Entscheidung nur sehr selten für die allgemeine Lösung. Warum?

Indikatoren:

1. Wie müssen Entscheidungen im Projekt gerechtfertigt werden? Gibt es eine Kultur des Minimalismus und der Feindlichkeit gegenüber Neuem? Gibt es Aussagen wie „Für unser Projekt ist das alles gar nicht nötig. Wir können eine spezielle Lösung basteln. Sie ist ja nur für einen Kunden oder nur für kurze Zeit."?

2. Wie ist das Verhalten gegenüber Kollegen, die sich „verschätzt" haben? Wird von Kollegen Hilfe geleistet? Gilt es als selbstverständlich, Wagnisse einzugehen? Vielleicht, weil es ausgezeichnete „externe" Kollegen gibt?

3. Wie ist die Qualität des vorhandenen Systems? Je höher diese ist, desto höher wird auch die Qualität der eigenen Entwicklungen sein (man kann ja nicht schlechter programmieren als die anderen).

4. Wie ist die Planung des Projektes? Vom Projektleiter erstellt oder gemeinsam? Wie ist die Kostenrechnung? Wird Risiko belohnt oder bestraft, falls Schwierigkeiten auftreten?

5. Kennen die Entwickler überhaupt verschiedene Lösungen? (Lässt sich abprüfen!)

Der Kreislauf der Exzellenz

Ausgezeichnete Entwickler erzeugen ausgezeichnete Entwickler.

Learning-on-the-job: noch immer *die* Art und Weise, wie das Wissen der Softwareentwicklung weitergegeben wird. Design-Patterns und Framework-Architekturen deuten hier eine Besserung an.

Was bedeutet das in der Projektpraxis? Es sind soziale Strukturen nötig, um die Wissensweitergabe zu ermöglichen.

- Das Problem muss erkannt und offengelegt werden.
- Die Wissensweitergabe muss als Aufgabe der Älteren definiert sein.
- Es muss Zeit dafür eingeplant sein.
- Die interne Ausbildung und der Einsatz muss in Richtung der Komplettheit gehen und darf nicht dem kurzfristigen Interesse eines Projektleiters untergeordnet werden (Spannungsfeld).
- Alarmzeichen: Richtlinien für Mitarbeiter beinhalten Passagen wie „Projektorientiertes Verhalten ist erwünscht". Heisst auf Klartext dass die Weiterbildung durch Bücher und Kurse nur insofern gewünscht ist, als sie direkt dem Ziel des Projektes dient. Grundsätzliche Probleme erfordern jedoch grundsätzliches Wissen zu ihrer Lösung. Solche Projekte weisen oft defizitäre Architekturen und Arbeitsweisen auf.

Beispiel für Ausbildungsrichtlinien: Ausbildung neuer Mitarbeiter im CC-Inet (Internet Competence Center der Systor AG) in Java. Wollen wir einen „Java-Programmierer", einen Java-GUI Spezialisten oder einen Softwareentwickler mit breitem Horizont?

Der Traum vom Reuse

Nichts wird so oft gefordert und so selten erreicht wie die Wiederverwendbarkeit von Software. Ist der Mangel an Reuse ein technisches oder soziales Phänomen? Welche Rolle spielen Abstraktionen dabei? Wie sorgt die soziale Gliederung der Software-Entwicklung dafür, dass Software nicht wiederverwendet werden kann? Wie ist die Selbstsicht der Applikationsprogrammierer? Wie ihre Aufgabenstellung?

Java-Beans und Human-Beans: Der Traum vom Modul.

(Thanks to Roald Dahl, The Big Friendly Giant.)

Nichts ist so alt in der Software-Entwicklung wie der Traum vom Modul. Das Software-Modul steht für Wiederverwendbarkeit, Parallelisierung der Arbeit, Unabhängigkeit gegenüber Änderungen anderswo. Anhand von Modularer Dekomposition werden Systeme verständlich gemacht und Gruppen gegründet und Aufgaben verteilt.

Aber worauf gründet sich der Glaube an Module?

1. Es gibt eine richtige Kategorisierung des Systems

2. Es gibt atomar richtiges Verhalten, kontextfrei und ohne state

3. Zusammenarbeit wird durch Modulgrenzen strukturiert und geleitet.

Die Wirklichkeit zeigt, dass beide Annahmen im Allgemeinen nicht zutreffen. Jeder Kategorisierung nach einem Prinzip lassen sich beliebige andere gegenüberstellen, die zu ihr im Widerspruch stehen, aber dennoch wichtige Aspekte des Systems erfassen. Und funktionale Dekomposition bis hin zu kleinesten „atomaren" Einheiten vergisst immer, dass z.B. der Wert von Information nicht nur atomar bestimmbar ist, sondern aus den Verbindungen der Informationen untereinander mitbestimmt wird.

In diesem Sinne kann sogar die sog. objektorientierte Herangehensweise als auf wackligen Füssen stehend angesehen werden.

Was die Zusammenarbeit betrifft (Parallelisierung, Vereinfachung), so entstehen gerade durch die Module und die damit verbundenen Kategorisierungen die Barrieren, die eine Gesamtsicht des Systems erschweren und oft zu endlosem Streit führen: „Warum unterstützt euer Modul dies und jenes nicht?"

Durch die soziale Gruppenbildung an den Grenzen von Modulen wird aus einem Mangel an Funktionalität ein politisches und soziales Problem.

Technische Architektur und soziale Gliederung.

Wieso stimmen die technischen mit den sozialen Gliederungen so auffällig überein? (Sogar die neuesten Komponentenmodelle wie Enterprise-Java-Beans definieren wieder verschiedene Rollen und Gruppen).

Was am Frameworkmodell war für die Applikationsleute unannehmbar? (Sie mussten während der Arbeit den Hut wechseln; je nachdem an was sie arbeiteteten, waren sie Systemgruppe oder Applikationsgruppe: Trennung von Rollen und Gruppen wurde abgelehnt.

Macht der Diagramme: technische und soziale Kategorienbildung:

Wieso brauchen wir in der EDV immer Bilder für Architekturen? Z.B: werden die oben genannten Strukturen der Softwarearchitektur als Kästchen gemalt – verdinglicht. Natürlich sind es keine „Kästchen", haben keine scharfen Ränder – sind letztlich überhaupt nichts „Wirkliches".

Der „Verdinglichungseffekt" hat – ähnlich wie die sprachliche Kategorienbildung – jedoch meist unmittelbare Auswirkungen: Aus den verdinglichten Kategorien werden durch unterschiedliche Aufteilung von Aufgaben sofort soziale Gliederungen in Form von Gruppen und Abteilungen abgeleitet. Der ursprüngliche gemeinsame Gegenstand – das System in seinem Lebenszyklus – verschwindet.

Vernetzte Probleme und geteilte Arbeit: „Ilities" sind keine Module.

In letzter Zeit sind die sog. „Ilities" als Problemfelder der Softwareentwicklung besonders hervorgehoben worden:

„Ilities are architectural properties that often can cut across different levels or components of a system‘s architecture to be properties of the whole" [OMA-NG]

(Interoperability, composability, evolvability, extensibility, tailorability, security, reliability, adaptability, survivability, afforadability, maintainability, understandability...)

Von welcher Art sind Kategorien wie Sicherheit, Zuverlässigkeit, Qualität, Performance, Wiederverwendbarkeit?

Sie haben gemeinsam, dass sie die gesamte Softwarearchitektur durchdringen. Sie können also nicht durch Interfaces einfach isoliert werden. Dadurch, dass sie nicht isoliert werden können, ist es auch schwierig, sie physisch auf „Module" abzubilden. Daraus folgt wiederum, dass die soziale Abbildung auf eine Rolle bzw. Gruppe schwierig ist.

Verkaffern

Wie kann man folgendes Phänomen erklären? Akademiker, teils mit Doktortitel gehen nach dem Studien in die Industrie und werden Softwareentwickler. Nach einigen Jahren praktischer Programmierung erscheinen sie

1. unfähig zu jeglicher softwaretechnischer Abstraktion

2. kleben an bisher ausgeübten Tätigkeiten und sind Neuem gegenüber feindich eingestellt

3. Erfassen Probleme nur noch in der Sprache ihres Systems bzw. in ihrer Programmiersprache.

Es sieht so aus, als als sei ihnen die softwaretechnische Realität, die sie bei Beginn ihrer praktische Tätigkeit kennengelernt haben, geradezu zur absoluten Wirklichkeit geworden, die gar keine Alternative mehr zulässt: Die Welt ist 8 Bit breit und 64 KB gross geworden.

Eine Erscheinung, die nur in Analogie zum Begriff des „Verkafferns" erklärt werden kann. Jemand ist von einer neuen Lebenswelt so gefangengenommen, dass er vorherige Sichtweisen und Fähigkeiten verliert und diese neue Lebenswelt als ausschliessliche, faktische und richtige ansieht.

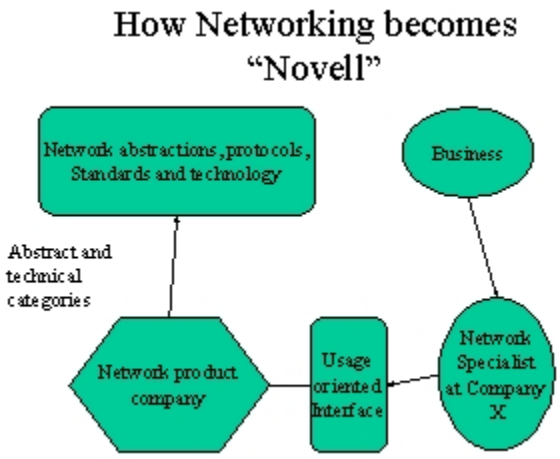
Weitere mögliche Einflussfaktoren sind:

1. Im Gegensatz zur abstrakten Ausbildung ist die Realität der Softwareentwicklung komplex. Auf diese Komplexität ist niemand vorbereitet, und man wird von ihr überwältigt.

2. Von abstrakter Ausbildung herkommend, sehen manche angesichts der Komplexität gar keine Möglichkeit der Abstraktion mehr. Software bleibt etwas, was man halt tut (mit der minimalen Ausrüstung des Erlernens einer Programmiersprache).

Das wirklich Schlimme daran ist, dass das Verkaffern keinesfalls nur Nicht-Informatiker betrifft. Selbst diejenigen mit Informatikausbildung zeigen dieses Verhalten.

Identität und Produkt oder warum Networking nicht Novell heisst.



Wie kann man folgendes Phänomen erklären? Bei vielen Spezialisten bekommt man das Gefühl, dass für sie ein bestimmtes Produkt, mit dem sie arbeiten, gleichbedeutend mit der Thematik geworden ist. Z.B. sieht ein Novell-Spezialist Networking als alles was Novell definiert, hat und kann. Der Spezialist verfügt nicht mehr über eine abstrakte Vorstellung von Networking. Teilweise bekommt man das Gefühl, dass die intime Kenntnis des Produktes einen guten Teil der Identität ausmachen.

Erklärungsversuche:

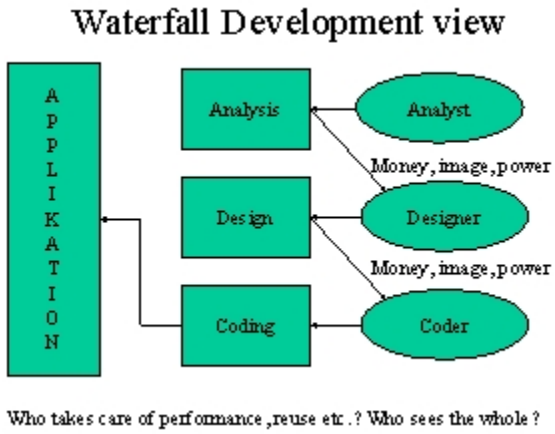
1. Wie im Fall des „Verkafferns" ein Mangel an Ausbildung, die auf die Komplexität der softwaretechnischen Realität vorbereitet.

2. Kein abstrakter Begriff von Themen praktisch erworben

3. Das „fette" Interface des Produktes überwältigt durch die Vielzahl von konkreten Einstellmöglichkeiten, ohne jedoch die dahinterliegenden Konzepte und Abstraktionen zu liefern. Dies ist mittlerweile auch in Ausbildungs- und Kursunterlagen als „bei Problem X erst F3 dann F5 drücken" eingegangen. Bücher gewisser Verlage (SAMS, QUE etc.) liefern Erklärungen in Form von Bildschirmausdrucken des GUIs.

Das Fatale an dieser Form des „Verkafferns" ist, dass es zunächst für die Firmen produktiv erscheint, wenn jemand nur durch „Knöpfchen drücken" Dinge wieder in Gang bringt. Auf die Dauer stellt sich jedoch heraus, dass eine kritische Sicht des Produktes nicht mehr möglich ist. Bei Wechsel des Produktes (natürlich gegen extremen Widerstand des Spezialisten) erlischt das Wissen der Mitarbeiter.

Woran das Wasserfallmodell wirklich scheitert



Wieso gilt heute das Wasserfallmodell für die Softwareentwicklung als gescheitert? Es scheint in seiner Trennung von Analyse, Design und Implementation doch die wesentlichen Schritte des Entwicklungsprozesses zu beinhalten. Es scheitert meiner Ansicht (W. K.) nach auch nicht nur technisch – d.h. es ist ihm technisch nichts immanent, was zu einem Scheitern der Entwicklung führen müsste – es scheitert vielmehr, weil es eine technische Strukturen mit sozialen Strukturen in sehr problematischer Weise verknüpft.

Begründung:

Die Basis für den Cocktail sozialtechischer Probleme wird mit der Arbeitsteilung geschaffen, fein unterteilt nach Analytikern, Designern und Implementatoren (gemeinhin „Kodierschweine" genannt. Dass dies keine nur horizontale Arbeitsteilung ist, zeigt sich darin, dass die Gehalts- und Prestigeverteilung rein vertikal ist: Analytiker bekommen am meisten Gehalt und Prestige. Bekommen Designer noch Visitenkarten?

Den einzelnen Arbeiten wird der Inhalt der auszuführenden Rolle genau festgeschrieben. Es entstehen Identitäten, die sich genau mit diesen Rollen identifizieren. An der Spitze ein Gefühl der Überlegenheit, verbunden mit einer Abscheu vor konkreten, „design- oder implementationspezifischen" Dingen, den Niederungen des Alltags. Am Ende das Gefühl derjenige zu sein, der den ganzen Unsinn der Analytiker und Designer letztlich ausbaden muss, bzw. durch eigene Vorstellungen so hinbiegen muss, dass das Produkt letztlich fertig wird. (Es sind Fälle bekannt, wo Implementationsgruppen am Ende ihr eigenes Design entwarfen, da das offizielle angeblich nicht zu gebrauchen war).

Für jede der drei Rollen werden eigene Wertsysteme, Rituale und Rechtfertigungen entwickelt. Irgenwie muss ja der soziale Unterschied gerechtfertigt werden. Für „die ganz unten" verstehen „die ganz oben" einfach nicht, dass die eigentliche Arbeit ganz unten geleistet wird und umgekehrt.

Und letztlich unterliegt dem Ganzen noch eine zeitliche Struktur der Form, dass Analyse vor Design und Design vor Implementation zu geschehen hat. Daraus ergibt sich die Notwendigkeit kompletten Wissens für das Wasserfallmodell. Wenn dieses komplette Wissen nicht auf allen Ebenen gegeben ist, dann sorgen die sozialen Schranken dafür, dass jede Korrektur den Anstrich von Fehler, Versagen hat. Rückmeldungen

von unten nach oben sind schwierig und immer gleich ein Politikum. Wissen über den Zusammenhang von Analyse, Design und Implementation entsteht nirgends.

Häufig werden die Arbeitsschritte auch noch an verschiedene Firmen vergeben. Damit wird eine klare Strukturierung der Verantwortlichkeiten angestrebt, im Regelfall entsteht jedoch an den Übergängen enorme politische Spannung. Wasserfallmodelle enden meist in Form des Fingerzeigens. Die wichtigste Tätigkeit in Wasserfallmodellen besteht darin alle Gespräche, Vereinbarungen etc. genau zu dokumentieren, um bei Bedarf Munition für die politische Auseinandersetzung zu haben.

Obwohl das Modell heute nicht mehr so oft angewandt wird, spielen sich ähnliche Szenen doch immer wieder zwischen Gruppen ab, die an einer gemeinsamen Entwicklung beteiligt sind. Opfer sind meist genau die technischen Probleme, die eine übergreifende Sichtweise gebraucht hätten. Nur ist die Strukturierung hier eine horizontale statt wie im Wasserfallmodell eine vertikale.

Wie kann vermieden werden, dass die vertikale Struktur des Vorgehens nach Analyse, Design und Implementation in eine ebenso vertikale soziale Struktur einmündet? Die Regel lautet dazu ganz einfach, dass wer eine Analyse erstellt, auch am Design und in der Implementation mitarbeitet. In der Praxis sind dies erfahrene Softwarearchitekten an deren Seite jüngere Kollegen und Kolleginnen an allen Phasen mitarbeiten. Die praktische Erfahrung aus der Implementation sorgt wiederum für den notwendigen Feedback, den auch der Architekt braucht.

Geschlechter

Wieso gibt es kaum Frauen in der Systementwicklung?

Wieso findet man mehr Frauen in der Modellierung (z.B. OOA, Datenmodellierung, SGML) oder in den analytischen Bereichen (Data-Mining, Data-Warehouse)?

Lassen sich die Ergebnisse der IIG Studie zum Studienanteil von Frauen in der Informatik auch damit in Zusammenhang bringen?

Wenn Projekte wachsen, schliesst sich der Kreis

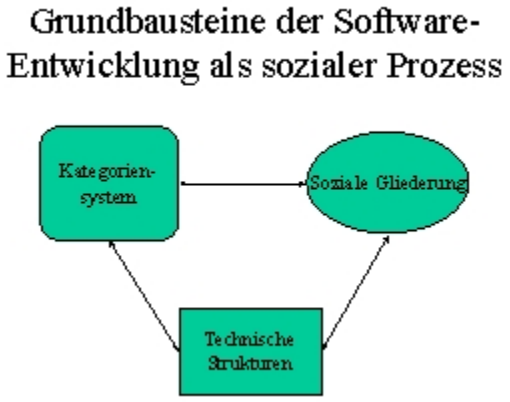
Am Anfang beruht die Kommunikation fast ausschliesslich auf informellen Beziehungen. Jeder kennt jeden. Allmählich entstehen neue Gruppen. Einzelne Mitarbeiter werden jetzt Vorgesetzte. Noch verlassen Mitarbeiter kaum das neue Projekt. Aussenkontakte entstehen (andere verwenden die Produkte des neuen Projektes). Mehr und mehr treten formelle Beziehungen auf, oft gegen den Widerstand der Entwickler eingeführt. Z.B. zieht das Projekt zurück an den Standord der Firma. Vorbei mit dem Sonderweg.

Viele „altgediente" Mitarbeiter des neuen Projektes empfinden dies als schlecht und verlassen die Firma. Kritikpunkte sind mangelnde Flexibilität, keine Dynamik mehr, Bürokratie etc. Man kann sagen, dass ein Gefühl grosser Enttäuschung entsteht, und die Kritikpunkte sind nicht weit weg von denen am alten Projekt.

Soziale Strukturen: Grösse von Gruppen, Abteilungen. Grad der Organisation. Grad der Arbeitsteilung. Die Rolle von Privilegien.

Chapter 6. ERKLäRUNGSVERSUCHE

Wir wollen drei Elemente zur Erklärung der oben gemachten Beobachtungen einsetzen und zwar in jeweils statischer wie auch dynamischer Sicht, d.h. über den Lebenszyklus eines Projektes oder Produktes hinweg.



1. Kategoriensystem

Darunter verstehen wir grob die auf Sprache und Bildern beruhenden Vorstellungen von Software, Architektur, Technik etc. Abstraktionen wie Prototypen aber auch Vorstellungen von Komplexität und Lebenszyklus – oder ihr Fehlen.

Zu diskutieren ist auch, wie wirklich diese Modelle und Kategorien sind Die Modelle der Informatik und EDV sind natürlich nicht Abbildungen von Realität. Sie sind verschiedene Sichten auf Phänomene. In ihren Auswirkungen – wenn sie zu Kategorien geworden sind – scheinen sie jedoch härter als Stein zu sein.

1. Soziale Strukturen

Die Einteilung auf Gruppen. Aufgaben und Rollen. Arbeitsteilige Lebenswelt. Identität und Kommunikationsstile bzw. Kompetenzen. Hierarchien in der Firmenorganisation aber auch Dauer der Zugehörigkeit. Arbeitsrechtliche Stellung (selbständig, angestellt). Die Ausprägung externer und interner Gliederungen z.B. Entwicklungen nach dem Wasserfallmodell mit streng hierarchischer interner Gliederung von Projektgruppen nach Architekten, Designern und Programmierern.

Neben diesen formellen Strukturen gibt es auch informelle Strukturen: z.B. Freundschaften und Seilschaften.

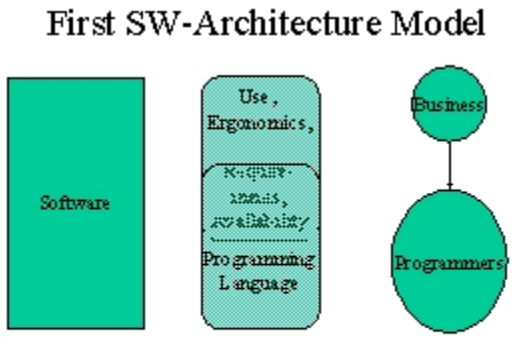
1. Technische Strukturen

Von der Mikroebene des Source-Codes über die Gliederungsweisen von Systemen (logisch, physisch), Architektur und dynamisches Laufzeitverhalten bis zur globalen Organisation von technischer Infrastruktur in Form von Enterprise Information Systems. (siehe [MOMA97])

Evolution technischer Architektur, sozialer Gliederung und ihrer Kategoriensysteme

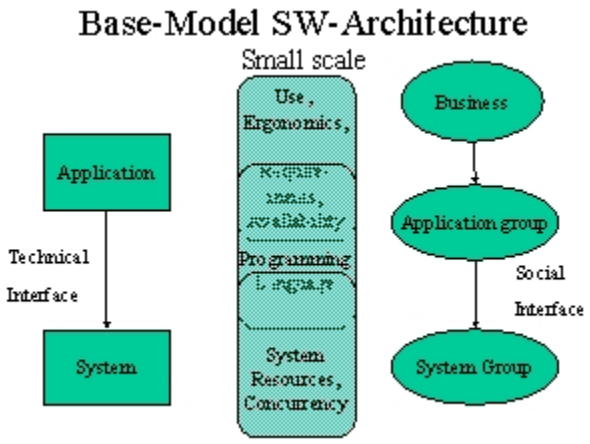
Ur-Modell

Das Ur-Modell der Softwarearchitektur ist gekennzeichnet durch die fehlende Dekomposition von Software und sozialer Gliederung. Heute findet man es noch teilweise im Bereich der Embedded-Control-Programmierung.



A piece of software gets downloaded to special hardware. It contains system and application. No decomposition of software or programming

Das Basis-Modell der Software-Architektur im kleinen Rahmen



Dieses Modell kann man als *den* Prototypen der letzten 20 Jahre Softwareentwicklung schlechthin bezeichnen. Die Trennung der Softwarearchitektur in System und Applikation spiegelt sich in der sozialen Organisation wieder. Es treten wesentliche Unterschiede in der sozialen Kommunikation innerhalb der Gruppen und zwischen ihnen auf. Die Inter-Gruppen-Kommunikation ist ebenfalls am „Uses"-Interface der technischen Interfaces orientiert.

Kennzeichen des sozialen Interfaces sind:

- Applikationsprogrammierer erwarten gewisse Funktionalitäten.
- Die Funktionalitäten müssen abgestimmt werden, typischerweise durch die beteiligten Projektleiter
- Die Interfaces sollen gleich bleiben, sonst entstehen Kosten durch Änderungen

- Die Systemfunktionalität muss zeitlich *vor* der Applikation vorhanden sein.
- Applikationsprogrammierer definieren ihre Rolle in Bezug auf die Business-Anforderungen und die Geschwindigkeit, mit der sie befriedigt werden können.
- Systemprogrammierer definieren ihre Aufgabe als „Schutz des Systems"

Dieses Modell und seine Interfaces macht bei Neuentwicklungen grosse Probleme, weil dort gegen diese Annahmen zunächst verstossen wird. Das Modell funktioniert, wenn es voll implementiert ist. Es ist keine Hilfe bei der Entwicklung eines solchen Systems. Änderungen sind sowohl technisch als auch politisch schwierig.

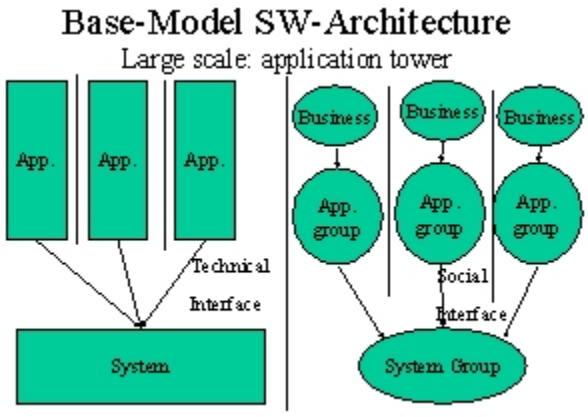
Das Basis-Modell der Software-Architektur im grossen Rahmen

Aus dem obigen Basis-Modell entwickelt sich im Laufe der Zeit das sog. Application-Tower-System. Es entstehen mehrere Applikationen in mehreren Gruppen. Die Entwicklungen laufen typischerweise separat und teilen kaum Wissen oder Software.

Fundamentale Änderungen der Business-Anforderungen bedeuten jetzt plötzlich die Änderung vieler Applikationen.

Das Interface zum System ändert sich technisch wie sozial. Die Systemgruppe ist konfrontiert mit einer Vielzahl technischer „Requests" aus verschiedenen Gruppen. Zwangsläufig muss auf der Systemebene jetzt generalisiert werden – die verschiedenen Einzelinteressen müssen kombiniert und implementiert werden. Das soziale Interface bedeutet den Umgang mit vielen Gruppen – im Gegensatz zur Applikationsentwicklung, die sich weiterhin an einer Business-Anforderung ausrichtet.

Dieses Modell ist in grossen Firmen vorherrschend, und fast alle Firmen versuchen sich davon zu befreien. Die Kosten sind sehr hoch, meist existiert ein enormes „Backlog" von unbefriedigten Business-Anforderungen (eigenartig: war doch gerade das eine der Rollendefinitionen der Applikationsentwicklung: die schnelle Befriedigung). Die Ursachen dafür liegen in enorm gewachsenen Applikationen, deren Wartung und Erweiterung schwierig ist, und die zudem über gemeinsamen low-level Datenzugriff in gegenseitiger Abhängigkeit stehen.

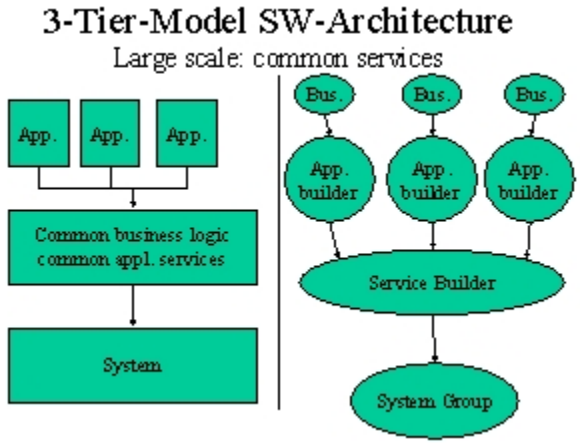


Der Three-Tier-Ansatz mit gemeinsamen Services

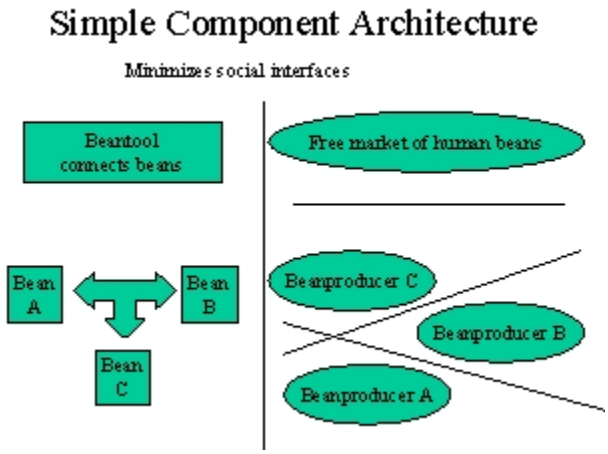
Ein Ausweg aus dem Dilemma der Applikation-Towers wird im Three-Tier-Modell mit Service-Layer gesehen.

Die Entwickler des Service-Layers stehen vor einem grossen Problem: An sie werden von oben Anforderungen in den unterschiedlichsten Sprachen und mit meist geringem Abstraktionsgrad herangetragen. Sie müssen jetzt quasi eine neue Sprache mit Abstraktionen definieren, die den gemeinsamen Anforderungen aller genügen.

Die Applikationsgruppen empfinden den Service-Layer als Konkurrenten, der Funktionalität aus ihrer Tätigkeit nimmt. Die Schnittstelle zum Service-Layer (sozial und technisch) wird schnell zum Politikum.



Einfaches Komponentenmodell: Java-Beans



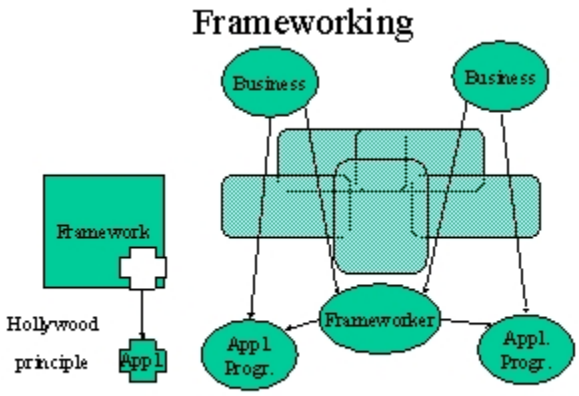
Dies ist der in der Einleitung erwähnte Traum vom Bau von Software durch Komposition und ohne Entwickler. Kennzeichen dieses Modells ist die radikale Reduktion des sozialen Interfaces auf ein Anbieter/Kunde-Verhältnis.

Das Modell stösst jedoch in dieser simplen Form auf technische Probleme: Die Definition der technischen Rollen (Interfaces, Implementationsweisen etc.) erfolgt nur minimal. Damit lassen sich nur manche allgemeine Funktionalitäten (z.B. ein Drucker-Bean) implementieren. Das Unterbringen orthogonaler Funktionalität (transactions, persistence, concurrency etc.) ist nicht möglich.

Deutlich wird jedoch bereits im einfachen Komponentenmodell die verstärkte Rolle von Tools. Java-Beans z.B. beruht im wesentlichen auf Namenskonventionen und Event-Handling. Beides Dinge, die ohne Toolsupport sozial schwierig zu handhaben sind, da sie Programmierer schnell überfordern, bzw. stumpfsinnig zu erstellen sind.

Framework mit Komponenten und Tools

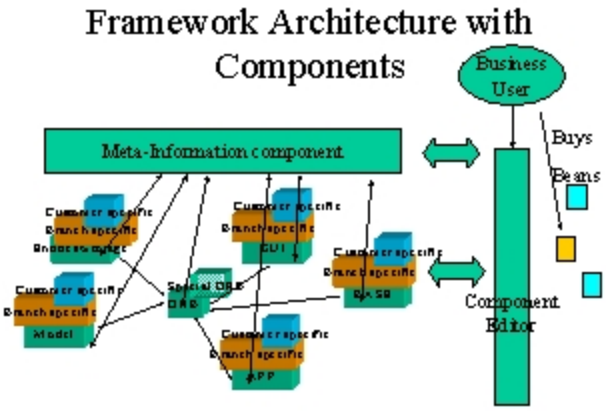
Die neuen Architekturen bringen völlig neue Ausrichtungen auch für die Rollen und Zielvorgaben der Softwareentwickler mit sich. Wichtigstes Merkmal ist, dass Applikationsprogrammierer quasi eine Stufe tiefer rutschen, d.h. allgemeinere Baublöcke zur Verfügung stellen. Sie entfernen sich damit auch aus der direkten, speziellen Ausrichtung auf *einen* Business-Auftraggeber und müssen daher lernen, ihr Kategoriensystem in Richtung grösserer Generalität umzubauen.



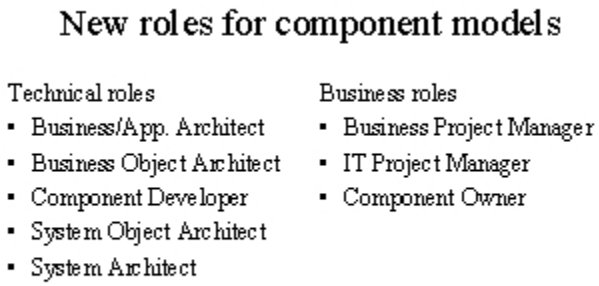
Was in der technischen Struktur so klar und logisch aussieht, verursacht auf der Seite der sozialen Strukturen erhebliche Probleme, da nun die Kategoriensysteme durcheinander geraten. Für den Applikationsentwickler steht die Welt jetzt Kopf, da sich der Frameworker zwischen Business und Applikation drängt. Eine Auflösung dieses Konfliktes könnte darin bestehen, dass Frameworker und Applikationsentwickler eine Personal- oder Gruppenidentität haben.

Ein weiterer wesentlicher Bestandteil ist die Möglichkeit des Einkaufs von Komponenten. D.h. interne Entwickler stehen in grösserer Konkurrenz zu externen Firmen.

Für die Business-User ist neu, dass ihnen die Möglichkeit gegeben wird, selbst aus vorhandenen Komponenten neue Applikationen zu erstellen.



Unterstützt werden muss ein solches Komponentenmodell durch eine neue Aufteilung der technischen und geschäftlichen Rollen:



Deutlich wird auch herausgehoben, dass der Konvertierungsaufwand zu einem solchen System einem Kulturwandel gleichkommt. Er muss durch Ausbildung, Neuorganisation, Prototypen, Referenzarchitekturen, Style- und Construction-Guides, Mentoring und Anfangsprojekte (Camps) abgefedert werden. Der finanzielle Aufwand solcher Lösungen ist gerade bei Grossfirmen beträchtlich und macht den Ankauf von fertigen Komponenten noch viel verlockender.

Eine weitere Erschwernis für in-house Entwicklungen ist, dass eine Organistation von Belohnungsformen für wiederverwendbare Komponenten nicht existiert. Dies lässt sich darauf zurückführen, dass es auch gar kein Controlling gibt, das diese Formen der Entwicklung versteht. Wie bewertet man finanziell eine Entwicklung die nicht mehr nur an der unmittelbaren Befriedigung eines speziellen Business-Wunsches ausgerichtet ist, sondern sich erst anschliessend – vielleicht in ganz anderen Gruppen und Projekten – finanziell auszahlt?

Wie erklärt man dem Management, dass es gerade die Ausrichtung auf spezielle Bedürfnisse und deren schnelle Befriedigung war, die zum Chaos der Application-Towers geführt hat?

Wesentlich für die technische Architektur ist, dass die Framework-Struktur durch ihre vorgegebenen Muster einen Teil der sozialen Auseinandersetzung um die Architektur von Software abnimmt. Tools, die mit dem Framework gekoppelt sind, sorgen für die Einbindung der entwickelten Komponenten und deren Anreicherung mit zusätzlicher Funktionalität. Diese Funktionalität ist nicht mehr vom Komponentenentwickler zu programmieren, sondern kann deskriptiv „gewünscht" werden.

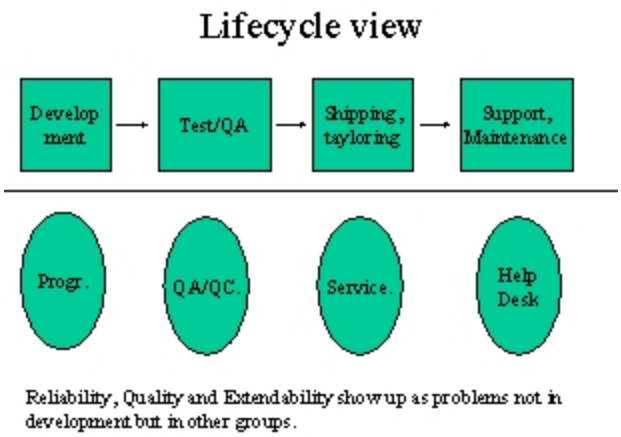
Beispiele dieser Architekturen:

- IBMs ComponentBroker (http://www.software.ibm.com/ad/cb)
- Enterprise Java Beans (http://www.java.sun.com/products/ejb)
- CORBA Components (http://www.omg.org)

Das Fehlschlagen der Transformation vom Base-Modell zum Component/Framework-Modell lässt sich auf dieser Grundlage klarer fassen. Neben den – zugegebenermassen nicht einfachen technischen Problemen – sind es soziale und kategoriale Probleme, die den Übergang erschweren:

- Das Rollenverständnis der Applikationsentwickler wird durch Frameworkentwickler bedroht. Diese benötigen und erlangen plötzlich ebenfalls Business Know-How.
- Die Möglichkeit fortgeschrittener Modellierung und deskriptiver Fassung von Business-Logik erlaubt Anwendern plötzlich, teilweise eigene Applikationen zu erstellen, oder vorhandene Applikationen anzupassen. Sie erwerben somit Fähigkeiten, die vorher den Applikationsentwicklern vorbehalten waren. Entsprechend schwierig ist es auch, die Applikationsentwickler dazu zu bewegen, Dinge nicht mehr direkt im Source-Code zu programmieren, sondern deskriptiv und damit explizit zu modellieren.
- Die Möglichkeit das Framework für Applikationsbedürfnisse selbst anzupassen stellt zunächst eine Erweiterung der Aufgaben und Kompetenzen der Applikationsentwickler dar. Leider wird sie im Allgemeinen nicht als solche wahrgenommen, da bis dato die Schnittstelle zum System als unveränderbar und gegeben gesehen bzw. erwartet wurde. Ausserdem bedeuten tiefere Anpassungen einen Übergang von spezieller (applikatorischer) Programmierung zu allgemein verwendbarer (systemnaher) Codierungsweise.

Technische und soziale Strukturen im Lebenszyklus eines Projektes



Reliability, Quality and Extendability show up as problems not in development but in other groups.

In der Praxis wenig beachtet wird die zeitliche Dimension eines Softwareprojekts. Dies wird begünstigt durch die Arbeitsteilung, die das Kategoriensystem aller Beteiligten formt. Dies führt dazu, dass die Aspekte nachgeordneter Gruppen in der Entwicklung wenig Beachtung finden – z.B. im Design überhaupt nicht auftauchen.

Beispiel: Was ist ein Softwareprodukt?

- Entwicklersicht: Funktional zusammenhängende logische und physische Entitäten.
- Servicesicht: Eine Anzahl von Files, die auf einem Rechner installiert werden müssen.

Problem: Der Entwickler kennt die Files, der Service nicht. Die Software muss über sich selbst Auskunft geben können, damit der Service seine Aufgabe erfüllen kann.

Neue Technologien und Modelle:

Weitere alternative technische Modelle sind im Entstehen.

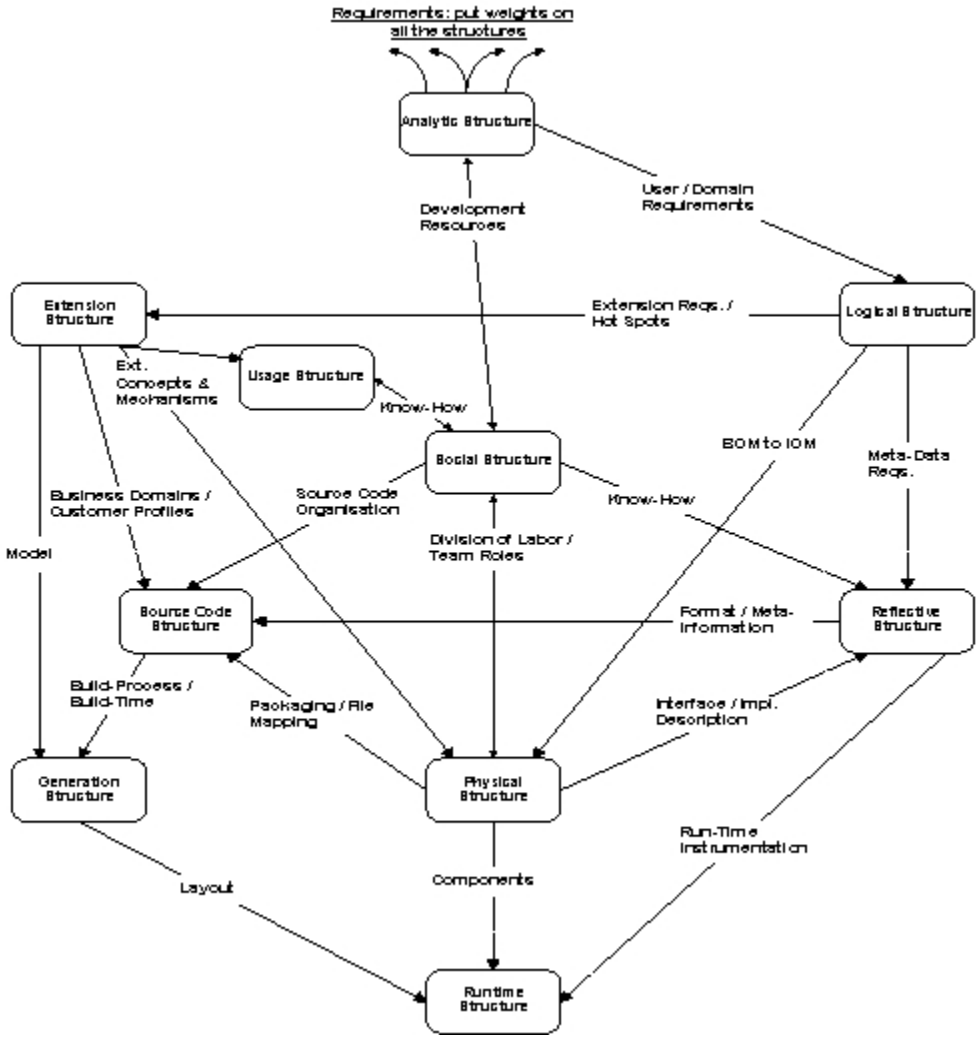
- Distributed Systems
- Open Implementations
- Meta-Object Protocols
- Subsumption Architectures
- Event-based Systems

Ihr Kennzeichen ist, dass sie die Rollen der herkömmlichen Architekturen verschwimmen lassen oder eventuell ganz auflösen. Jemand kann eine Applikation programmieren und über die Open-Implementation-Schnittstellen gleichzeitig am Kernsystem Änderungen vornehmen, die seine Applikation wesentlich erleichtern.

Während wir uns langsam an das Verschwinden von Applikationen als zentrales Gliederungsmittel von Softwarelösungen gewöhnen, bringen verteilte Systeme letztlich das Verhalten von Software selbst in eine neue Dimension: Verhalten ist nicht mehr prozedural-deterministisch, sondern wird zum emergenten Phänomen zur Laufzeit durch die Beteiligung vieler Rechner (Kevin Kelly, Out of Control...).

Das sind ganz klare Anschläge auf das Selbstverständnis von Softwareentwicklern: Lösungen, die nur durch Ausprobieren oder evolutionär entwickelt werden können. Sie werden die sozialen und kategorialen Grenzen neu definieren.

Chapter 7. VERWOBENHEIT VON SOZIALEN UND TECHNISCHEN STRUKTUREN



Some requirements of successful projects

- Multi-dimensional decomposition of architecture
- Projection of technical and social architecture over time
- Make category systems explicit: no single "right" view
- Expect multiple and changing category systems. Architecture must support those
- Beware of "mapping" approaches. They try to reduce complexity to just one category system and fail in reality
- Minimize social interaction using framework technology
- Maximize social interaction by separating social interfaces from technical interfaces
- Micro level of coding: Make the connection between complexity and abstraction visible and socially understandable.

Voraussetzungen erfolgreicher Projekte:

1. Multidimensionale Dekomposition der Architektur

Dies ist ein klares Bekenntnis dazu, dass es nicht „eine richtige" Sichtweise des Systems gibt. Verhindert werden soll die gewaltsame Reduktion eines komplexen Zusammenhangs auf einige wenige Kategorien denn dies führt im Allgemeinen dazu, dass wesentliche Teile „vergessen" werden. Entscheidungen über den Grad der Ausführung von Strukturen werden explizit getroffen. Der Zusammenhang der Architektur geht nicht verloren.

1. Projektion der technischen und sozialen Architektur über die Zeitachse.

Technische und soziale Anforderungen an die Architektur werden explizit gemacht. Nachgeordnete soziale Rollen und Aufgaben finden sich durch die Architektur unterstützt.

1. Kategoriensysteme werden als solche explizit gemacht.

Es wird davon ausgegangen, dass es kein „richtiges" Kategoriensystem gibt, sondern dass die Sichtweise auf die Architektur völlig unterschiedlichen Aspekten unterliegen kann.

1. Mehrere, sich ändernde Kategoriensysteme.

Die Architektur muss den Einsatz verschiedener Kategoriensysteme sowie deren Wandel unterstützen. Dies ist besonders ein Problem der Objekttechnologie, die sich am objektivistischen Klassenkonzept orientiert.

1. Vorsicht vor „Mapping"-Konzepten

Scheinbar unverzichtbares Handwerkszeug der Software-Architektur ist das sog. Mapping. Business-Funktionen „mappen" zu technischen Services. Objekte „mappen" die Realität. Analyseergebnisse „mappen" zu Design und Implementation (Oft unterstützt durch Code-Generatoren). Typisch für dieses Vorgehen ist die gewaltsame Reduktion der Komplexität auf ein Kategoriensystem. Es bewirkt, dass andere – gleichermassen gültige – Gliederungsweisen unterdrückt werden. Entscheidende Unterschiede zwischen z.B. menschlicher Abstraktion und Maschine verschwinden.

Beispiel: Wir kennen einen Begriff (Prototypen) von Tischen. Davon auszugehen, dass unser Denken deshalb auch in Form von Tischobjekten geschieht, ist Unsinn. Genau dieser Schluss wurde und wird in der Objekttechnologie nach wie vor praktiziert.

1. Minimierung sozialer Interaktion durch Framework-Technologie

Wo technisch möglich, sollte das Verhalten und das Modell des Systems durch das Framework vorgegeben sein. Dies reduziert die sozialen Auseinandersetzungen um technische Funktionalität enorm und bietet darüberhinaus einen kategorialen Rahmen der Architektur, der ihr Erlernen leichter macht.

1. Maximierung sozialer Interaktion, wo kein Framework möglich ist.

D.h. die technischen Schnittstellen dürfen nicht gleich zur sozialen Gliederung laufen. Die soziale Gliederung nach Gruppen und Rollen soll sich mit den technischen Interfaces überschneiden. Dies verhindert die übermässige Ausbildung nur eines technischen und eines sozialen Kategoriensystems.

1. Mikroebene der Softwareentwicklung

Durch Coding-Standards und Konstruktionsrichtlinien werden die Strukturen der Architektur sozial vermittelbar. Coding-Standards verbinden darüber hinaus die Ebene der Komplexität (sichtbar an der Zeile Source-Code, die gerade geschrieben wird) mit der Ebene der Abstraktion (den Strukturen der Architektur, die durch diese Zeile gerade berührt werden). Da die Ebene der Abstraktion beim Programmieren nicht sichtbar ist, müssen Programmiersprache und Coding-Standards Hinweise darauf geben.

Chapter 8. LITERATUR

[OMG-NG], Craig Thompson, Ted Linden, Bob Filman, Thoughts on OMA-NG – The next Generation Object Management Architecture, 1997

http://www.omg.org/CORBA/greenpaper.html

[LAKOFF87], George Lakoff, Women, Fire and Dangerous Things – What Categories Reveal about the Mind, 1987, Chicago University Press

[GABRIEL96], Richard Gabriel, Patterns of Software – Tales from the Software Community, Oxford University Press, 1996

[KPK97] Dany Kesch, Stefan Plüss, Walter Kriha, Architectural Structures for Large Scale Systems. OOPSLA Workshop Paper

http://www.ccsi.com/~brr/LargeSysInfo.html

[RKR96] Richard K. Runyan, Parts – The Newest Cottage Industry. WOON96 Sankt Petersburg, 1996

[KELLY94], Kevin Kelly, Out of Control – The New Biology of Machines, Social Systems and the Economic World, Addison-Wesley, 1994

[WHORF56] Benjamin Lee Whorf, Language, Thought and Reality: Selected Writings of Benjamin Lee Whorf, MIT Press Cambridge MA, 1956

[MOMA97], Mowbray-Malveau, CORBA Design Patterns 1997