

# Systematische Softwareentwicklung

(C)opyright Ralf Bürger 2001-2003

**"Viele möchten in einem Tag verschlingen, was sie kaum im ganzen Leben verdauen können"**  
(Balthasar Gracián)

## Inhalt

1	Einführung.....	3
1.1	Vorwort .....	3
1.2	Zielgruppe .....	4
1.3	Konventionen .....	5
1.4	Änderungen .....	6
2	Vorgehensweise .....	7
2.1	Systematik .....	7
2.2	Modelle .....	8
2.3	Meine Vorgehensweise .....	9
2.4	UML .....	10
2.5	Wozu MS-Project? .....	11
3	Anforderungen .....	13
3.1	Funktionale Anforderungen .....	13
3.2	Technische Anforderungen .....	13
3.3	Anforderungen fixieren .....	14
4	Pre-Analyse .....	15
4.1	Projektmappe.....	15
4.2	Verantwortung.....	15
4.3	Name und Hauptziel.....	15
4.4	Zweck und Tragweite (engl. "Purpose & Scope") .....	16
4.5	Prosatext .....	17
4.6	Was also ist die Pre-Analyse? .....	18
4.7	Angebote und Verträge .....	19
5	Analyse.....	21
5.1	Analyseaspekte.....	21
5.2	In/Out-Liste .....	22
5.3	Interviews .....	23
5.4	Problemzerlegung.....	24
5.5	Analysemuster (engl. analysis patterns).....	25
5.6	Anwendungsfälle.....	26
5.7	Spezifikationsmuster .....	26
5.8	Prototyp .....	29
5.9	Testfälle.....	29
5.10	Dokumente .....	30
5.11	Gesamt-Anwendungsfalldiagramm.....	30
6	Design .....	32

6.1	Realisierung.....	32
6.2	Kick-Off-Meeting.....	32
6.3	Architekturen.....	33
6.4	Design der Modelle.....	34
6.5	Design Patterns.....	37
6.6	Software Ergonomie.....	37
7	Implementierung.....	38
7.1	Implementierungsaspekte.....	38
7.2	Codierung.....	38
7.3	Leitung.....	39
7.4	Iterationen.....	39
7.5	Änderungsanforderungen.....	40
8	Test.....	42
8.1	Warum testen?.....	42
8.2	Testen bis zum jüngsten Tag.....	42
8.3	Weiche Tests.....	42
8.4	Systematische Tests.....	43
8.5	Qualität.....	43
9	Übergang.....	45
9.1	Übergang der Software an den Kunden.....	45
10	Wartung.....	46
10.1	Wartung.....	46
11	Verweise.....	47
11.1	HTTP-Links.....	47
11.2	Bücher.....	47

# 1 Einführung

## 1.1 Vorwort

**"Ich sage, was ich denke, damit ich höre, was ich weiss."**  
(Unbekannt)

Ich habe im Laufe der letzten 20 Jahre schon viele Projekte und viel Software gemacht und sehr viel mit Kunden, Projektmitarbeitern und Kursteilnehmern über die "richtige" Vorgehensweise bei der Softwareentwicklung diskutiert. Nun fange ich doch tatsächlich auch mal damit an, meine Erfahrungen und meine Meinung aufzuschreiben. Es gibt zwar schon so einige Bücher zur Softwareentwicklung, aber die Branche ist noch immer sehr jung und es gibt noch viel Handlungsbedarf zur Weiterentwicklung der Systematik.

Mein persönlicher Schwerpunkt liegt eindeutig bei den ersten Phasen eines Software-Projekts - der Rest ist für mich nur technische Realisierung, die sich im Laufe der Jahre zunehmend vereinfacht hat und in Zukunft auch recht stark automatisiert werden wird. Wenn erst einmal klar ist, was wie gemacht werden soll, dann ist die Umsetzung kein unüberwindbares Problem mehr - daher sind die ersten Projektphasen einfach die wichtigsten.

Ein Mini-Beispiel für all das, was ich hier schreibe, ist dieses kleine [Hütchenspiel](#).

Momentan ist dieses Dokument noch in großen Teilen eine Stichwortsammlung, aber der Umfang nimmt (fast) täglich zu. Feedback heiße ich jederzeit sehr willkommen, also scheuen Sie sich bitte nicht: [eMail@RalfBuerger.de](mailto:eMail@RalfBuerger.de)

rabu - 23.12.2001

## 1.2 Zielgruppe

**"Mit dem PC lösen wir die Probleme, die wir ohne nicht hätten."**

(Unbekannt)

Wer sollte diese Abhandlung lesen? Nun, momentan tun Sie es! Das ist auf jeden Fall schon einmal gut, denn Sie scheinen sich für Softwareentwicklung zu interessieren. Welche Aufgabe Sie bei der Softwareentwicklung übernehmen, ist dabei egal, denn ich versuche, den gesamten Entwicklungsprozess abzudecken.

Es ist keinesfalls erforderlich, die gesamte Abhandlung auf einmal zu lesen - oder überhaupt ganz zu lesen -, denn ich bemühe mich sehr kompakt zu schreiben und möglichst viel für Sie hinein zu packen. Ich habe halt nicht vor, ein möglichst dickes Buch zu schreiben, damit ich es möglichst teuer verkaufen kann. Lesen Sie einfach mal hinein, nehmen Sie einen Tipp auf, setzen ihn um und schon sind Sie ein Stück weiter gekommen (falls nicht, lassen Sie es mich bitte wissen, damit ich den Tipp ändern kann). Diese Arbeitsweise ist in der Regel effizienter, als 1000 Seiten auf einmal zu lesen oder ins Regal zu stellen. Wenn Sie aber ganz ganz ganz viel lesen möchten, kaufen Sie alles, was Sie an Büchern unter [Verweise](#) finden, lesen das alles durch, setzen es in die Praxis um und schicken mir dann ein paar weitere Tipps ;-)

### 1.3 Konventionen

Mit "Anwender" meine ich natürlich auch immer "Anwenderin", ich sollte also eigentlich besser immer "**AnwenderIn**" schreiben. Dies ist aber dann im weiteren Verlauf auch wieder schwierig, weil es dann immer "sie/er" oder "ihr/ihm" heißen muss. Da ich hier schon genug mit der deutschen Grammatik an sich und der neuen Rechtschreibung im Besonderen zu kämpfen habe, gestatten Sie mir bitte, dass ich immer nur in einer Form formulieren muss. Da ich es in meinem Job leider fast nur mit Männern zu tun habe, fiel meine Entscheidung auch zugunsten der männlichen Form. Also "Anwender" - basta!

Wenn etwas wie auf Papier gedruckt, nach Quelltext oder wie an Command-Line eingetippt aussehen soll, sehen Sie diese nichtproportionale Schreibmaschinen-**Courier**-Schriftart (sofern Ihr Browser macht, was ich will).

## 1.4 Änderungen

**"Beständig ist nur der Wechsel."**

(mein Physiklehrer)

Hier die letzten Änderungen in dieser Abhandlung (neueste oben, älteste unten), damit Sie nicht immer wieder alles lesen müssen:

[Konventionen](#) erweitert

Und noch [ein](#), [zwei](#), [drei](#), [vier](#), [fünf](#) Sprüche

Kapitel [Systematik](#) überarbeitet

Und wieder ein [Spruch](#)

Zweites [Pre-Analyse-Blatt](#)

Navigationsbaum im linken Frame jetzt ohne Links auf den Kapiteln, nur noch auf den Seiten

Kapitelangabe im Balken oben auf den Einzelseiten

Weitere Sprüche oben auf den Einzelseiten

Umstellung auf Einzelseiten

Subdomain sse.RalfBuerger.de etabliert

Navigationsbaum im linken Frame

[Inhaltsverzeichnis](#) um Kurzzangaben ergänzt

Jedes Kapitel hat jetzt oben auf der Seite einen - meines Erachtens - passenden Spruch

Link [L17](#) und Absatz zu [SPICE](#) ergänzt

Kapitel [Übergang](#) endlich mal angefangen

Buch [B21](#) ergänzt

[Datenmodell](#) erweitert um Normalisierung

Neues Diagramm in "Meine Vorgehensweise"

Datenmodell neu

Spezifikationsmuster fertig

Problemzerlegung fertig

Zielgruppe neu

Änderungsanforderungen neu

Download und Konventionen neu

Design erweitert um MVC-Architektur

Analyse erweitert

n-tier-Modell überarbeitet

Kapitel Anforderungen erweitert

n-tier-Modell neu

Weitere Beispiele für MS-Project

Analyseaspekte angefangen

Modelle erläutert, Prosatext überarbeitet und Verantwortung eingefügt.

neue Beispiele für Zweck- und Tragweiten-Diagramme

Ein Vorwort gibt es nun auch

Pre-Analyse fertiggestellt

Pre-Analyse-Blatt neu

Inhaltsseite zu "Systematische Softwareentwicklung" (so heißt es jetzt)

Vorgehensweise: Ein aktuelles Problem

Pre-Analyse: Name und Hauptziel

Vorgehensweise: Sichten-Diagramm

Pre-Analyse: Diagramme und Textänderungen

Vorgehensweise: großes USDP-Diagramm und MS-Project-Screenshot

## 2 Vorgehensweise

### 2.1 Systematik

**"Künstliche Intelligenz ist kein Schutz vor natürlicher Dummheit."**

(Unbekannter)

Im März 1967 erschien im Magazin "Fortune" ein Artikel über den für damalige Begriffe neuartigen Prozess der "Softwareentwicklung". Ein Auszug daraus: "In der Programmierung herrscht nicht annähernd die Disziplin wie beispielsweise in den Naturwissenschaften. Deshalb spielt die Intuition eine große Rolle. Noch unterscheiden sich die Programmierer in ihren kreativen und intuitiven Fähigkeiten." Im Jahre 1968 prägten Softwareprofis auf einer NATO-Konferenz zur Softwareentwicklung den Ausdruck "Softwarekrise" und beschrieben damit Schwierigkeiten bei der Erstellung zuverlässiger Systeme. Ist die Softwarekrise vorüber?

Probleme auch aus jüngster Zeit zeigen, dass dies offensichtlich noch immer nicht der Fall ist:

#### **Raumstation ISS nach Computerpanne ins Trudeln geraten**

4. Feb. 2002

Die Internationale Raumstation ISS ist nach einem Computerproblem im russischen Teil des Komplexes ins Trudeln geraten. Für sechs Stunden konnte die Station nicht korrekt gesteuert werden, was zu Problemen bei der Stromversorgung führte, da die Solarzellen nicht auf die Sonne ausgerichtet waren.

Wie die Nasa am Montag mitteilte, konnte das Problem von der russischen Bodenkontrolle schließlich gelöst werden. Der fehlerhafte russische Computer, der die Ausrichtung der ISS im All kontrolliert, wurde neu gestartet und arbeitete danach wieder normal, wie es hieß.

Text: dpa

Bildmaterial: Nasa



Die Zeitschrift [CIO](#) schreibt in der Ausgabe 6/2002: "Ein Viertel aller Projekte scheitert, die Hälfte hält Zeit- und Budget-Rahmen nicht ein. In den regelmäßigen Befragungen der [Standish Group](#) haben sich diese Zahlen bis heute kaum verändert." Am 25.03.2003 gibt die Standish Group bekannt, dass die Fertigstellung der Projekte auf 34% gestiegen ist, eine Verdopplung gegenüber 16% in 1994. Andererseits sind die Projekte mit Zeitüberschreitungen auf 82% gestiegen, gegenüber 63% im Jahr 2000. Und während 2000 noch 67% der angeforderten Funktionalitäten im endgültigen Produkt enthalten waren, sind es jetzt nur noch 52%.

Woran liegt das? Es ist sehr schwer, die reale Welt im Computer mit Software nachzubauen, weil dabei alle relevanten Gesetze der realen Welt nachempfunden werden müssen. Das ist stets deutlich aufwändiger, als man zunächst denkt. Wenn man dann noch mit den Schwierigkeiten durch ständige Änderungen an den bei der Entwicklung benutzten Systemen und Tools sowie gänzlich neuen Technologien kämpfen muss und dann noch Probleme im Team auftreten, ist das Projekt kaum noch zu retten. Dann kommen noch die Fehlerbehebungen hinzu und all die Änderungen, die der Kunde noch wünscht. Und wenn man dann vermeintlich fertig ist, weiss man ziemlich genau, welcher unglaublich große Aufwand eigentlich noch ins Projekt gesteckt werden müsste, um alles wirklich wasserdicht zu bekommen. Und dafür hat man nie die benötigte Zeit und das benötigte Geld, weil bis dahin sowieso schon Zeitplan und Budget überschritten sind. Also wird es nicht zeitig fertig, enthält nicht alles, was man wollte, aber dafür wurde es teurer und hat viele Fehler!

Das Wesentliche dabei dürfte die Unterschätzung der Komplexität der realen Welt sein. Wenn beispielsweise eine Verwaltungssoftware eines Schlachthofs das Gewicht eines Schlachtviehs über das Wochenende von 450 kg auf 300 kg reduziert hat, so weiß diese Software offensichtlich zwar, dass durch Verdunstung das Gewicht eines Schlachtviehs in der Halle über zwei Nächte am Wochenende stärker abnimmt als über eine Nacht innerhalb der Woche, aber diese Software hat offensichtlich kein Gespür dafür, dass die Reduzierung von 450 kg auf 300 kg bei nur einer weiteren Nacht (und selbst bedingt durch einen Feiertag über zwei weitere Nächte) völlig "unrealistisch" ist. Dieses "Gespür" müsste man eigentlich in Form von teilweise komplexen Plausibilitätsprüfungen an sehr vielen Stellen implementieren, womit sich der Aufwand aber nochmal unerwartet erhöht - zumindest für den Entwickler unerwartet, denn der Mitarbeiter der Firma sagt zu dem ganzen Problem sowieso nur: "Das war doch wohl klar, habt Ihr das etwa nicht berücksichtigt? Das weiss man doch wohl!" (Ich bin diesem Problem übrigens wirklich 1994 bei einem türkischen Fleischgroßhandel in Köln begegnet.)

Aber warum kann eine Werft ein 200 Meter langes Kreuzfahrtschiff mit zwölf Decks für 1200 Passagiere und 400 Personen Besatzung rechtzeitig, fehlerfrei, zum vereinbarten Preis und mit allen geforderten Leistungen liefern, und warum geht das bei Software offensichtlich nicht? Ist es schwieriger Software zu schreiben als ein Kreuzfahrtschiff zu bauen? Oder liegt es daran, dass wenige Spezialfirmen wenige Kreuzfahrtschiffe bauen und an jeder Ecke selbsternannte Softwareentwickler sich mit selbst überlegten Methoden an zu großen Softwareprojekten versuchen? Auch wenn man den Aufwand für die Entwicklung gigantisch hochschraubt und tatsächlich keine Rücksicht auf Zeit und Geld nehmen muss, wird Software nie fehlerfrei! Dieses haben unzählige wirklich große unternehmenskritische Projekte gezeigt. Liegt es vielleicht doch daran, dass man Software nicht anfassen kann und mehrere Entwickler gemeinsam an einem virtuellen Gebilde arbeiten, während man ein Kreuzfahrtschiff modular aus realen Bausteinen zusammensetzt? Kann man tatsächlich das Problem der Software-Fehler auf folgende einfache Formel reduzieren: "Software hat Fehler, weil sie aus Logik besteht." [B4].

Was können wir tun, um bei der Softwareentwicklung Fehler zu vermeiden oder zumindest möglichst früh zu finden und die Entwicklung von Software und damit die Software selbst robuster zu gestalten? Und wie können wir Software-Projekte vorhersehbarer, kalkulierbarer und damit planbarer gestalten? Betrachten wir doch mal die Definition für "Software-Engineering" (deutsch: "Software-Technik") vom ANSI-Komitee [L14]: "The systematic approach to the development, operation, maintenance and requirement of software." (deutsch: "Der systematische Ansatz für die Entwicklung, den Betrieb, die Wartung und die Anforderung von Software.")

Ich denke, die meisten Softwareentwickler sollten überhaupt erst mal die erprobten und bewährten systematischen Ansätze kennen lernen und auch anwenden. Das würde vielen Projekten und damit auch den Statistiken schon mal sehr gut tun. Denn die einfacheren Projekte werden naturgemäß auch von den unerfahreneren Leuten abgewickelt, die sich noch gar nicht mit Systematischer Softwareentwicklung beschäftigt haben und deshalb auch nicht professionell mit diesem "virtuellen Gebilde" umgehen können. Die großen, wirklich komplexen Projekte scheitern wohl wirklich daran, dass es in just diesen Projekten immer noch zu wenig verfügbare Erfahrung, zu wenig Standards, zu wenig anerkannte Schnittstellen für die Umsetzung des jeweiligen Stücks der realen Welt gibt. Es gibt beispielsweise noch nicht so viele internationale Flughäfen mit vollautomatisiertem Gepäckbeförderungssystem und erst recht gibt es noch nicht viele Entwickler, die dieses komplexe Projekt schon mehrfach abgewickelt haben. Auch die Technologien, die dort zum Einsatz kommen, sind teilweise noch keine 15 Jahre alt. Die wirklichen Professionals können meines Erachtens nur durch viele viele Projekte der gleichen Komplexität sowie eine evolutionäre Verbesserung der bestehenden, angewandten, systematischen Ansätze dem Ziel näher kommen, ein Softwareentwicklungsprojekt rechtzeitig, im Kostenrahmen, fehlerfrei, vollständig und zur Zufriedenheit des Kunden abzuschließen!

## 2.2 Modelle

Betrachten wir dazu einige Modelle, nach denen Software entwickelt wurde bzw. wird. Einige Modelle zur Softwareentwicklung in ihrer historischen Abfolge sind: [Wasserfall-Modell](#) - [V-Modell](#) - [Spiral-Modell](#) - [Prototypen/Casey-Modell](#) - [USDP](#)).

Im Prinzip war das Wasserfallmodell gar nicht so verkehrt, denn es ist sicherlich immer gut, einen Schritt nach dem anderen zu tun. Ein ganzes Software-Projekt strikt stufig durchzuführen geht aber

schief. Wie ist es denn bei einem echten Wasserfall? Am Ende wird man doch nass, oder? Das Problem beim Wasserfallmodell ist, dass man am Anfang noch gar nicht alle Eventualitäten abschätzen kann, während der Entwicklungszeit viele neue Änderungsanforderungen (Change Requests) eintreten und am Ende weder zeitlich ("**in time**") noch preislich ("**in budget**") noch qualitativ ("**in quality**") das herauskommt, was sich der Auftraggeber vorgestellt hat. Damit ist der Ärger *vorprogrammiert*.

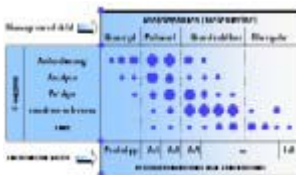
Zerteilt man das Software-Projekt in viele Phasen, durchläuft jede Phase stufig und bezieht den Kunden (die Anwender) in jeder Phase mit ein, so sieht es schon viel besser aus, weil jede einzelne Phase abschätzbar wird und Änderungen nach jeder einzelnen Phase in die Planungen integriert und getestet werden können. Diese Vorgehensweise wird heute als "**iterativ inkrementell**" bezeichnet, weil das Produkt in vielen schleifenähnlichen Durchläufen von Version zu Version vollständiger wird.

Bedenken Sie immer, dass das eigentliche Ziel der Softwareentwicklung nicht die Programmierung selbst, sondern die Auslieferung einer fertigen Software mit den drei oben genannten Teilzielen ist.

USDP

Die Modelle USDP ("Unified Software Development Process") bzw. RUP ("Rational Unified Prozess") gehen auf die drei Amigos Grady Booch, Jim Rumbaugh und Ivar Jacobson von der Firma [Rational Software \[L7\]](#) zurück. In Deutschland ist die Firma [oose \[L4\]](#) ("OOSE" = Object Oriented Software Engineering) rund um Bernd Oesterreich sehr aktiv in der Verbreitung des objektorientierten Ansatzes.

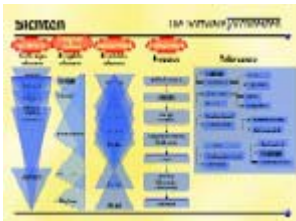
Beim USDP-Modell wird zwischen der Managementsicht und der technischen Sicht unterschieden: Die Managementsicht besteht aus Projektphasen mit Meilensteinen für Konzept, Entwurf, Konstruktion und Übergabe. Die technische Sicht besteht aus Prototypen und Produktversionen. Bei jeder Versionsstufe werde die Prozesse Anforderung, Analyse, Design, Implementierung und Test durchlaufen (**iterative** Vorgehensweise). Wie bereits oben erwähnt wächst dadurch das Produkt **inkrementell** in seinem Funktionsumfang.



[großes USDP-Diagramm \(115 kb\)](#)

Es wird dabei gerne auch vom "**Lebenszyklus**" des Software-Entwicklungsprojektes gesprochen (engl. "Software Development Life Cycle"). Die vier Phasen werden auch als Etablierung (engl. "Inception"), Ausarbeitung (engl. "Elaboration"), Konstruktion (engl. "Construction") und Übergang (engl. "Transition") bezeichnet.

Ich meine, dass dieses Modell um weitere **Sichten** (Rollen) des Projekts ergänzt werden sollte, denn neben der technischen Versionssicht und der Management-Phasensicht gibt es mindestens noch eine Produktsicht, die für Marketing, QS, Training und Support relevant ist, sowie eine kaufmännische Sicht (Verträge, Zahlungen und Strafen), die für den Vertrieb sehr wichtig ist.



[großes Sichten-Diagramm \(399 kb\)](#)

Sie können an der Breite der Dreiecke sehen, welche Prozesse wie intensiv iteriert werden. So reicht die Konzeptphase der Projektleitersicht beispielsweise vom Anforderungsprozess bis zum Designprozess, wobei ein Schwerpunkt auf dem Anforderungsprozess liegt. Die Version 1.0 der Marketingsicht hingegen hat ihren Schwerpunkt bei der Übergabe.

## 2.3 Meine Vorgehensweise

**„Es ist ein Vorteil im Leben, die Fehler, aus denen man lernen kann, frühzeitig zu machen“**

(Sir Winston Churchill)

Wie Sie schon an der Struktur dieser Site sehen, setze ich im Vertriebsgespräch mit einer Diskussion der [Anforderungen](#) an und biete dem Kunden eine kurze kostenpflichtige [Pre-Analyse](#) an. Diese ergibt ein grobes Bild über den zu betreibenden Aufwand, liefert mir einen ersten Einblick in das Unternehmen des Kunden und führt zu der Entscheidung über die Machbarkeit. Am Ende der Pre-Analyse-Phase steht eine Grobspezifikation (früher "Lastenheft" genannt) des zu erstellenden Systems.

Auch die [Analyse](#) biete ich in aller Regel zum Festpreis an, so dass sich der Kunde über den Aufwand möglichst klar ist. Die Analyse ist per Definition unvollständig, schafft aber genug Sicherheit für ein präzises Angebot, für eine Zeitabschätzung, für einen Releaseplan der Funktionsblöcke und für die Skizzierung des geplanten Produkts. Häufig soll am Ende der Analysephase auch eine vollständige Feinspezifikation (früher "Pflichtenheft" genannt) des zu erstellenden Systems stehen, also eine Beschreibung darüber, wie das System exakt aussehen wird und was es wie genau leisten wird. Oft macht eine vorläufige Aufwandsabschätzung nach der Hälfte der Analysezeit Sinn. Nach der Analysephase kenne ich jedenfalls definitiv die Fachlichkeit des Kunden und kann die Widerspruchsfreiheit der Anforderungen gewährleisten. Dadurch, dass der Kunde bei der Analyse sehr viel Zeit für Interviews zur Verfügung stellt, kann auch die Vollständigkeit und fachliche Fehlerfreiheit gewährleistet werden.

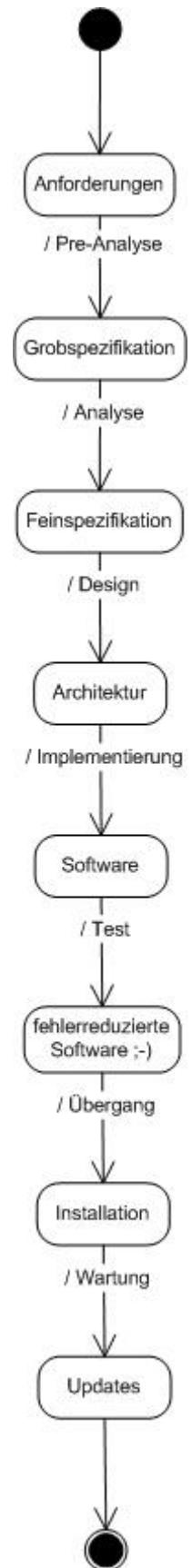
In der [Design](#)phase erfolgt das technische Design der Architektur der Software, d.h. die Klassenhierarchie, das ER-Modell, die Verteilung der Komponenten innerhalb der Server-Farm und schließlich auch die Struktur der Clients. Der Designer gewährleistet die Implementierbarkeit der erstellten Analyse und Spezifikation. Die [Implementierung](#) ist schließlich die Umsetzung in die physikalische Technik, d.h. die Generierung der Tabellen in der Datenbank, die Programmierung mit der gewählten Sprache und die Gestaltung des GUI (Graphical User Interface, deutsch: Graphische Benutzerschnittstelle).

[Tests](#) erfolgen immer, wirklich andauernd! Bereits mit dem Zielsatz kann der Name der Software auf seinen Bezug getestet werden, mit der Pre-Analyse wird das Fachkonzept getestet, mit der Analyse die ersten Strukturen aus der Pre-Analyse, etc. etc. etc. Der letzte Test erfolgt nach dem [Übergang](#) der Software in das Echtssystem des Kunden durch die Anwender. Die bei der Abnahme anhand des Analysedokuments festgestellten und im Laufe des Betriebs auftretenden Mängel werden unverzüglich in der [Wartungs](#)phase behoben. Hier werden auch in Folgeversionen die aufgeschobenen Wünsche realisiert.

Ein wesentlicher Vorteil der iterativ inkrementellen Arbeitsweise wird von Siebel in [\[B19\]](#) sehr schön mit diesem Spruch ausgedrückt: "Get it in the water now - launch and learn; it's better to float something than to have a perfect vessel in drydock" (dt. "Bring es nun ins Wasser - starte und lerne; es ist besser irgendwas zu fluten als ein perfektes Schiff im Trockendock zu haben"). Das soll nicht heißen, dass man jeden Quark ungetestet auf den Markt werfen soll, denn das sind dann "banana products" (das Produkt reift beim Kunden), aber man soll auch nicht warten bis das Produkt zwar perfekt, aber dafür veraltet ist und die Firma zwischendurch vielleicht schon ausgehungert ist.

## 2.4 UML

Die Unified Modeling Language ([UML \[L11\]](#)) wurde von der Object Management Group ([OMG \[L3\]](#)) am 17.11.1997 in der Version 1.1 als Standard verabschiedet. Die UML bietet Möglichkeiten, um von der ersten Aufnahme eines Geschäftsprozesses über die Analyse von Anwendungsfällen bis zum Entwurf von Klassendiagrammen lückenlos graphisch arbeiten zu können. "Zur Zeit sprechen alle Anzeichen dafür, dass die UML die Notation der Zukunft für die Objektorientierung wird." [\[B7\]](#)



Falls Sie statt der Online-Spezifikation ein echtes Buch aus Papier bevorzugen, empfehle ich [B2] und [B13]. Die UML hängt immer stark mit den "drei Amigos" der Firma Rational ([L7]) zusammen, weil diese den Vorschlag über mehrere Jahre ausgearbeitet und bei der OMG zur Standardisierung eingereicht haben.

## 2.5 Wozu MS-Project?

Microsoft Project ist eine Software für das PC-gestützte Projektmanagement. Sie können Vorgänge zeitlich festlegen und zu Sammelvorgängen zusammen fassen. Darüber hinaus können Sie Ressourcen (Mitarbeiter) zuordnen und über hinterlegte Stundensätze Kosten berechnen. Abhängigkeiten zwischen den Vorgängen sorgen für automatische Verschiebungen ganzer Vorgangsketten bei unerwarteten Ereignissen. Mit Meilensteinen können Sie wesentliche Zeitpunkte des Projekts definieren und deren Einhaltung bzw. Überschreitung prüfen.

In verschiedenen Ansichten können Sie beispielsweise überlastete Mitarbeiter ermitteln oder Kostenverteilungen darstellen. Letztendlich können Sie MS-Project über die integrierte Programmiersprache [VBA](#) (Visual Basic for Applications) auch um eigene Funktionalitäten erweitern. Weitere Infos zum Produkt MS-Project erhalten Sie direkt von [Microsoft](#) [L15].

In einem Software-Projekt dient MS-Project immer wieder zur Beruhigung des Managements, das in der Regel keinen genügenden technischen Hintergrund besitzt, um sich selbst ein Bild über den Projektstand zu machen.



[großer MS-Project-Screenshot 1 \(407 kb\)](#)



[großer MS-Project-Screenshot 2 \(589 kb\)](#)



[großer MS-Project-Screenshot 3 \(637 kb\)](#)



[großer MS-Project-Screenshot 4 \(295 kb\)](#)

Denken Sie bei all der Begeisterung über die Möglichkeiten von MS-Project daran, das ein Tool nicht die Erfahrung eines Projektleiters ersetzen kann! Hier greift einer meiner Lieblingsprüche: "A fool with a tool remains a fool!" (dt. "Ein Dummer mit einem Werkzeug bleibt noch immer ein Dummer!").

Nutzen Sie MS-Project beim **ersten** Mal nur zur nachträglichen Dokumentation Ihres Projekts, beim **zweiten** Mal zur Protokollierung der aktuellen Geschehnisse und erst beim **dritten** Mal zur aktiven Kontrolle bzw. für Vorhersagen. Danach werden Sie MS-Project als Planungsinstrument in Ihren Projekten nicht mehr missen wollen.

Über das Management von Terminen, Ressourcen und Kosten hinaus sollten Sie sich die folgenden offenen Fragen stellen, um einen Eindruck von Ihrem Projekt zu bekommen:

Wann wird dieses Projekt fertig sein?

Was kostet dieses Projekt?

Worin besteht die Leistung dieses Projekts?

Welche Prioritäten gibt es in diesem Projekt?

Wer ist der Auftraggeber dieses Projekts?

Welche Rolle spiele ich in diesem Projekt?

Wer ist alles an diesem Projekt beteiligt?

Welche Bedeutung hat dieses Projekt für die Beteiligten?

Worin unterscheidet sich dieses Projekt von den anderen?

Sie werden feststellen, dass jedes Projekt, jeder Kunde und jeder Mitarbeiter anders ist, vor allem bei Software-Projekten. Und dies ist gut so, denn gerade das macht den Reiz bei Software-Projekten aus - es sein denn, Sie befinden sich in einem "[Scheissprojekt](#)" [[L10](#)] ;-)

## 3 Anforderungen

### 3.1 Funktionale Anforderungen

**"Das Schicksal mischt die Karten und wir spielen."**

(Arthur Schopenhauer)

Im Erstgespräch mit dem Kunden wird zunächst über Wünsche und Vorstellungen gesprochen, die in aller Regel noch nicht konkretisiert oder zumindest noch nicht abschätzbar sind. Meistens ist sich der Kunde sehr unsicher über Realisierbarkeit und Aufwand. Als Auftraggeber wird der Kunde zum Anforderungssteller und liefert in aller Regel ein Fachkonzept mit den funktionalen Anforderungen (engl.: functional requirements). Dies geschieht in irgendeiner Form, also als protokolliertes Gespräch, als Handschrift-Zeichnung, als unstrukturierter oder auch strukturierter Text oder vielleicht sogar schon als Software-Lastenheft.

Oft denkt der Kunde leider, dass bereits genügend Informationen zur Entwicklung der Software vorliegen, aber davon darf sich der Anbieter auf keinen Fall beeindrucken lassen, weil dies wirklich praktisch nie so ist. Der Anbieter sollte hier den Kunden unbedingt über die weitere Vorgehensweise aufklären und erläutern, welcher immenser Detaillierungsgrad noch erarbeitet werden muss. Es gibt halt nur zwei Möglichkeiten: entweder lernt der Kunde, wie professionell Software entwickelt wird oder der Anbieter lernt die Fachlichkeit und die Geschäftsprozesse des Kunden; in aller Regel ist Letzteres einfacher und billiger!

Sollte das Fachkonzept bereits als fertige Spezifikation aller Details ausgeprägt sein (beispielsweise bei Ausschreibungen auf Basis bereits erstellter Analysen), kann direkt eine Aufwandsabschätzung folgen. Legt der Kunde eine Anforderungsliste vor, sollte diese in der Analyse auf Vollständigkeit, Widerspruchsfreiheit und Implementierbarkeit geprüft und anhand eines Fragenkatalogs abgerundet werden. Meistens enthält das Fachkonzept aber so viele Lücken und Unklarheiten, dass in der Analysephase die wirklichen Anforderungen erst noch erarbeitet werden müssen. Hier sollte auch darauf geachtet werden, ob der Anforderungskatalog mehr einer Weihnachts-Wunschliste entspricht, in der alles enthalten ist, was man schon immer mal haben wollte, aber nicht sinnvoll abgegrenzt wurde, was wirklich in der ersten Stufe implementiert werden sollte. Hier sollte dann die Regel "Reduce to the Max" ("Reduziere auf das Maximum") befolgt werden.

Beim Analyseprozess muss genau ein definierter Ansprechpartner des Kunden mit repräsentativen Vertretern aller Endanwendergruppen (Akteure) aktiv mitarbeiten, um auch wirklich den echten Bedarf zu ergründen und nicht an den Bedürfnissen der Anwender vorbei zu entwickeln. Nur so können die während der Entwicklung auftretenden gefürchteten Änderungsanforderungen (englisch "change requests") vermieden werden.

Die Zeitschrift [CIO](#) schreibt in ihrer Ausgabe 6/2002: "Caper Jones, Chef-Forscher beim PM-Software-Anbieter Artemis, hat herausgefunden, dass sich in 90 Prozent aller Fälle die Anforderungen während der Projektlaufzeit verändern. Um dieses Problem zu begrenzen, rät er, IT-Lösungen gemeinsam mit den Endbenutzern zu entwickeln, wodurch sich die Hälfte der Änderungswünsche eliminieren lasse. Prototypen könnten weitere 10 bis 25 Prozent der Änderungen in eine frühe Phase verlagern. Für große Projekte empfiehlt Jones so genannte Change Control Boards aus Management, Benutzerrepräsentanten und Entwicklern."

### 3.2 Technische Anforderungen

**"Who in their right mind would ever need more than 640K of RAM?"**

(dt.: "Welcher vernünftige Mensch würde jemals mehr als 640K RAM benötigen?")

(Bill Gates, Microsoft Gründer, 1981)

Die technischen Anforderungen stellt ebenfalls der Kunde und sie werden als Kapitel in die Anforderungsanalyse integriert. Dabei ist auf Grenzwerte und Plausibilitäten genauso zu achten wie auf Leistungsanforderungen (performance requirements), z.B. Übertragungsraten von Standorten oder Ant-

wortzeiten einer Datenbank. Die **Technologieplattform** (Soft- und Hardware) wird hier festgelegt und die Testlandschaft beschrieben.

Natürlich sollte der Anbieter auch bei den technischen Anforderungen dem Kunden mit **Rat und Tat** zur Seite stehen, aber er sollte auf keinen Fall den Kunden zu einer Technologie überreden, nur weil er diese selbst am besten beherrscht. In der Regel hat der Kunde auch bereits hohe Investitionen in der technischen Infrastruktur gebunden, so dass die Flexibilität hier eher gering ist.

Genauso sollte ein professioneller Softwareentwickler in der Lage sein auf die Betriebssystem-Neigungen des Kunden einzugehen und die momentan so beliebte **Religions-Diskussion** um UNIX-Derivat/MS-Windows oder Sun-ONE/MS-.NET/IBM-Websphere sachlich zu führen!

### 3.3 Anforderungen fixieren

**"I think there's a world market for maybe five computers."**

(dt.: "Ich denke, es gibt einen Weltmarkt für vielleicht fünf Computer.")

(Thomas Watson, IBM chairman, 1943)

Ganz wichtig ist, dass die gestellten Anforderungen **fixiert** werden, d.h. als Anlage ins Angebot der Realisierung genommen werden. Denn wenn es am Ende des Projekts um die Abnahme und Bezahlung geht, muss gegen diese geforderte Leistung getestet werden. Sehen Sie es mal anders herum: Wenn Sie nicht wissen, was gefordert wurde, wie wollen Sie dann wissen, was bezahlt werden muss und wie wollen Sie überhaupt wissen, wann Ihr Projekt zu Ende ist?

Dabei sollte grundsätzlich beachtet werden, dass Spezifikationslücken einen Spielraum für den Entwickler darstellen: Wird nur angegeben, **was** realisiert werden muss und nicht, **wie** es realisiert werden muss, kann sich der Entwickler den einfachsten Weg aussuchen. Dieser Weg wird dann sicherlich die fachlichen Anforderungen des Kunden erfüllen, aber vielleicht nicht gerade die Variante repräsentieren, die sich der Kunde immer vorgestellt - aber leider nicht spezifiziert - hat.

## 4 Pre-Analyse

### 4.1 Projektmappe

**"Kunde sein verdirbt den Charakter."**

(Ein Kunde von uns, den ich hier lieber nicht nenne ;-)

Bereits bei der geringsten Wahrscheinlichkeit für einen Auftrag sollte ein Projektordner mit Deckblatt angelegt werden. Darin werden auch die ersten Notizen des ersten Vertriebsgesprächs abgeheftet (und sei es ein bekritzelter Bierdeckel). Im weiteren Projektverlauf wird alles, wirklich **ALLES**, in diesem Ordner abgeheftet. Idealerweise gibt es im Projekt kein relevantes Papier außerhalb dieses Ordners!

Das Deckblatt sollte Angaben wie beispielsweise Projektnummer, Projektname, Projektleiter sowie Anfangs- und Enddatum des Projekts enthalten. Natürlich gehören auch Kundenangaben wie beispielsweise der Ansprechpartner dazu.

Kopien von Angeboten, Aufträgen und Rechnungen **können** in den Projektordner - Analysen, Spezifikationen, Architekturbeschreibung und Protokolle **müssen** in den Projektordner.

### 4.2 Verantwortung

"Nichts auf der Welt ist so gerecht verteilt wie der Verstand: Jeder glaubt, er hätte genug davon."

(Rene Descartes, frz. Mathematiker u. Philosoph, 1596-1650)

Die Pre-Analyse ergibt ein grobes Bild über den zu betreibenden Aufwand, liefert einen ersten Einblick in das Unternehmen des Kunden und führt zu der Entscheidung über die Machbarkeit. Außerdem lernen Kunde und Anbieter sich persönlich kennen, was der erste Schritt zum Aufbauen von Vertrauen ist.

Der Kunde hat bei Pre-Analyse und Analyse als Auftraggeber die Verantwortung für die Korrektheit und Vollständigkeit der Fachlichkeit, der Auftragnehmer hat als systematisch arbeitender Software-Experte hingegen die Verantwortung für die Konsistenz und Implementierbarkeit. Ferner muss der Auftragnehmer den Kunden darüber aufklären, was überhaupt möglich und sinnvoll ist. Manchmal denkt der Kunde, dass die eine oder andere Implementierung doch sehr einfach sein müsste, und ist überrascht, wenn nachher ein immenser Aufwand angeboten wird. Andererseits wagt der Kunde oft gar nicht, nach Funktionen zu fragen, deren Implementierung wirklich sehr einfach wäre.

Bei der Pre-Analyse wird der Grundstein für die Analyse und damit für das gesamte Projekt gelegt. Ganz wichtig ist es daher, von vorn herein die Verbindlichkeit, d.h. Schriftlichkeit, zu pflegen. Alle Informationen sollten dabei kurz, klar und konkret formuliert werden.

### 4.3 Name und Hauptziel

"Der ziellose Mensch erleidet sein Schicksal, der zielbewusste gestaltet es."

(Immanuel Kant)

Zuerst sollte der Name (ggf. Arbeitstitel) für das Projekt gefunden werden. Dieser sollte kurz und prägnant sein. Wenn es sich um ein Produkt handelt, das im Markt vertrieben werden soll, so ist natürlich zu prüfen, ob der Name nicht bereits geschützt ist. Ferner sollte ggf. geprüft werden, ob der Name in andere Sprachen übersetzbar ist oder in anderen Sprachen oder Ländern eine völlig andere Bedeutung hat.

Danach muss das zu erreichende Hauptziel des Projekts formuliert werden. Dies ist die schwierige Antwort auf die einfache Frage: "Wofür ist die neue Software gedacht?" Es ist ganz wichtig, dass das Hauptziel des Projekts in einem einzigen Satz formuliert wird. Sie werden sich wundern, wie oft Sie diesen Satz noch verwenden werden! Ob Sie in der Kantine von Kollegen nach ihrer momentanen Aufgabe gefragt werden, dem Marketing die Bedeutung des Produkts erläutern sollen, eine Produktbe-

schreibung für eine Messe suchen, einen Einstieg für ein Handbuch brauchen oder nach der Analyse noch mal schauen wollen, ob Sie überhaupt noch auf dem richtigen Weg sind: dieser eine Satz hilft Ihnen!

Ziele sollten immer SMART sein: Specific (spezifisch), Measurable (messbar), Achievable (erreichbar), Realistic (realistisch) and Time-related (zeitbezogen).

Beispiele:

Zu entwickeln ist eine Software, mit der die Kunden (Mitarbeiter) schneller und effizienter unterstützt und alle Störungen und Anfragen chronologisch vom Support oder Fachbereich erfasst, über eine Wissensdatenbank ausgewertet und nachgehalten werden können.

Zu entwickeln ist eine Software, mit der alle Bar-Einnahmen und -Ausgaben in einem Kassenbericht, der mit der Kasse an jemand anderes übergeben werden kann, zusammengefasst und nachgehalten werden können.

Das Hauptziel des Systems ist eine Konsolidierung der wesentlichen Daten aus den einzelnen Projekten zum Zweck der Steuerung, Dokumentation und Erfüllung des Berichtswesens bei der Abwicklung der Investitionspläne von *Firma* unter wirtschaftlichen Gesichtspunkten.

Ziel des Projektes "LUDOWare" ist die Entwicklung einer Software zur Unterstützung einer Spieleausleihe inklusive einer Mitglieder- und Spielverwaltung und einem Internetzugriff.

Ziel des Projektes "PHW-OWL" ist die Schaffung eines Internet-Portals für Hochschulen und Unternehmen aus dem Raum OWL, welches Praktika, Themen für akademische Arbeiten und Stellen vermittelt.

Es soll ein professionelles Management-Werkzeug namens "*Firma*-inventar" für die zentrale Hard- und Softwareinventarisierung mit Zuordnung zu Räumen und Personen erstellt werden.

#### 4.4 Zweck und Tragweite (engl. "Purpose & Scope")

"Sobald der Geist auf ein Ziel gerichtet ist, kommt ihm vieles entgegen."  
(Johann Wolfgang von Goethe)

Wenn Sie als Analytiker vom Kunden das Hauptziel bekommen haben, fragen Sie nach allen Leuten, die an der Definition dieses Hauptziels beteiligt waren. Es erscheint unglaublich, aber ich habe schon erlebt, dass acht(!) Leute aus verschiedenen Bereichen ein halbes Jahr für die Definition dieses einen Satzes gebraucht haben! In einem solchen Fall sollten Sie diese acht Leute auch alle das gemeinsam gefundene Ziel unterschreiben lassen! Über die Diskussion um die Beteiligten am Zielsatz finden Sie die ersten Hauptakteure und Interessensgruppen.

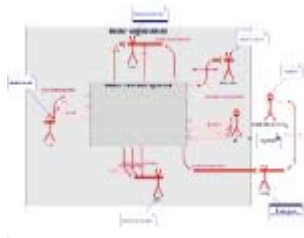
Die Akteure (engl. "actors") des neu zu erstellenden softwareintensiven Systems sind die Unternehmensrollen (z.B. "Buchhalter"), die später direkt mit dem System arbeiten werden. Wenn Sie zu viele Akteure finden (mehr als 7), sollten Sie die Hauptakteure (die wichtigsten Akteure) (engl. "main actors") bestimmen.

Die Interessensgruppen (engl. "stakeholders") bedienen das System nicht direkt, haben aber bestimmte Interessen daran, dass die Akteure dies tun.

Sprechen Sie mit den Akteuren und Interessensgruppen und nehmen deren Hauptanforderungen auf. Diese werden in der späteren Analyse weiter zerlegt und mit den Informationen aus dem Fachkonzept weiter gefüllt. Dadurch führen die Hauptanforderungen Sie auch zu den Teilzielen des Systems.

Beschreiben Sie mit den Hauptanforderungen was das System leisten soll, noch nicht wie es das tun soll, denn das System selbst ist in diesem Stadium noch eine Unbekannte (engl. "black box").

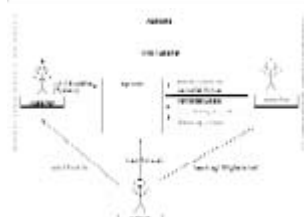
Zeichnen Sie ein "Zweck- und Tragweiten-Diagramm" als besondere Form des UML-Anwendungsfalldiagramms. Dabei zeichnen Sie das zu erstellende Software-System als innere Systemgrenze und die Unternehmensgrenze als äußere Systemgrenze. Nun können Sie die gefundenen Hauptakteure mit ihren Hauptanforderungen an das zu erstellende Software-System einzeichnen. Spätestens jetzt wird allen Beteiligten deutlich, wofür das zu erstellende Software-System eigentlich gut ist ("Zweck") und wen es alles betrifft ("Tragweite").



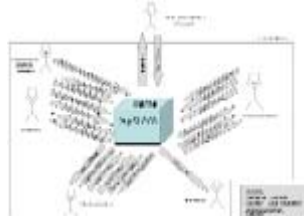
[großes Zweck- und Tragweiten-Diagramm 1 \(203 kb\)](#)



[großes Zweck- und Tragweiten-Diagramm 2 \(122 kb\)](#)

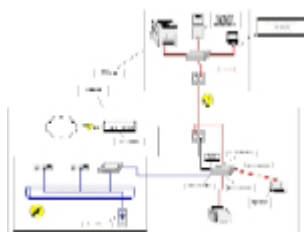


[großes Zweck- und Tragweiten-Diagramm 3 \(81 kb\)](#)



[großes Zweck- und Tragweiten-Diagramm 4 \(295 kb\)](#)

Dieses Verfahren zum Aufnehmen von Anforderungen kann natürlich nicht nur für die Erstellung von softwareintensiven Systemen genutzt werden: wer sind wohl die Hauptakteure von folgendem kleinen Netzwerk und welche Hauptanforderungen wurden bzw. werden wohl an dieses Netzwerk gestellt?



[großer Netzplan \(181 kb\)](#)

## 4.5 Prosatext

Beschreiben Sie nun in Prosa mit wenigen(!) kurzen und einfachen Sätzen die gefundenen Hauptanforderungen und damit die wesentlichen Teilziele des zu erstellenden Software-Systems. Dies ist sehr wichtig, weil dies die **Sprache** des Kunden ist (nicht jeder Kunde versteht UML-Diagramme mit Systemgrenzen und Strichmännchen!). Der Kunde kann an diesem Text das Verständnis seiner eigenen Fachlichkeit durch den Anbieter testen.

Denken Sie immer daran, dass der Kunde und der Anbieter verschiedene Sprachen sprechen: der Kunde denkt an seine Fachlichkeit und Organisation, der Anbieter denkt an Datenstrukturen und Modulschnittstellen. Unter dem Begriff "Ordner" beispielsweise verstehen der Kunde und der Anbieter oft ganz verschiedene Dinge: der Kunde meint den Ordner, in dem er mit Trennblättern Papier abheftet, den er auf eine definierte Art und Weise beschriftet und der in einem bestimmten Schrank unter der Obhut eines Mitarbeiters oder einer Mitarbeiterin verwahrt wird. Der Anbieter meint den Ordner, den er in einem Festplattenvolumen des PC-Netzwerks anlegt und mit Rechten eines Benutzers seines technischen Systems versieht, der darin Dateien speichern darf. Beides ist vielleicht nur die unterschiedliche **Sicht** auf den Informationsspeicher des Unternehmens und meinen damit letztlich das Gleiche, aber sind sich wirklich der Kunde und der Anbieter darüber im Klaren?

Der Prosatext ist auch ein Test für das Ziel- und Tragweiten-Diagramm, das formulierte Hauptziel sowie den Namen des neuen Softwaresystems. Wenn diese vier Sichten nicht übereinstimmen, muss entsprechend korrigiert werden.

#### 4.6 Was also ist die Pre-Analyse?

Die Pre-Analyse dient also dazu, festzustellen, in welcher Liga das Projekt spielt und welche Bedeutung es für den Kunden hat. Nach der Pre-Analyse können bereits die ersten Aussagen zu Machbarkeiten, Risiken, groben Aufwänden und den Kosten der Analyse getroffen werden.

Kern der Pre-Analyse sollte genau ein Blatt Papier sein, das die drei Elemente "Name und Hauptziel", "Zweck- und Tragweiten-Diagramm" und "Prosatext" enthält:

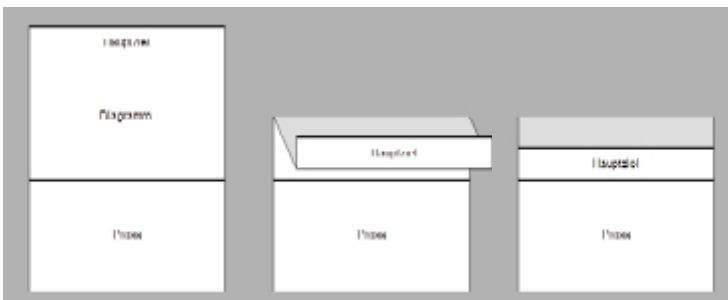


[großes Bild des Pre-Analyse-Blatts \(72 kb\)](#)



[großes Bild des fertigen Word-Pre-Analyse-Blatts \(64 kb\)](#)

Da ich mir zuerst in den Interviews beim Kunden Notizen mache, dann später daraus das Diagramm erstelle und zuletzt den Prosatext schreibe, kennt der Kunde maximal meine Notizen. In einer abschließenden Diskussionsrunde mit allen Hauptakteuren lege ich jedem Teilnehmer das Blatt wie folgt gefaltet vor:



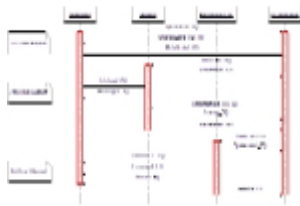
[großes Bild des Faltvorgangs \(72 kb\)](#)

Nachdem wir uns das Ziel in Erinnerung gerufen haben, lesen wir gemeinsam den kurzen Prosatext und steigen in die Diskussion ein. Dafür klappen wir das Blatt dann auseinander, weil das Diagramm gegenüber dem Text den großen Vorteil hat, dass ein Einstieg an jedem Punkt möglich ist. Meine Erfahrung mit dem Diagramm ist bei dieser Vorgehensweise durchweg positiv, weil der Kunde einerseits bereits am Text gemerkt hat, dass ich ihn ernst nehme und verstanden habe und andererseits jeder Kunde das Diagramm nach nur kurzen Erklärungen auch versteht.

#### Geschäftsprozesse

Oft sind den Unternehmen die **Prozesse** besser bekannt als die Beteiligungen oder gar Anforderungen einzelner Rollen im Unternehmen. Dies liegt wohl auch daran, dass in den letzten Jahren Heerscharen von Beratern "Prozessoptimierungen" gepredigt haben. Daher kann es für die Unternehmen verständlicher sein, mit den Prozessen anzufangen und daraus dann die Hauptakteure und deren Hauptanforderungen abzuleiten. Bei großen Systemen oder bei

vielen Prozessen wird sich dies schnell als der schwierigere Ansatz erweisen; wenn das zu erstellende Software-System aber hauptsächlich einen einzelnen Geschäftsprozess abdecken soll, ist es durchaus sinnvoll, im ersten Schritt diesen Prozess zu analysieren. Betrachten Sie als Beispiel den Mitarbeiter-Support, der in dem folgenden Sequenzdiagramm beschrieben ist (IV\_HelpDesk ist dabei das zu entwickelnde System):



[großes Sequenz-Diagramm \(191 kb\)](#)

Sie sehen in den ersten drei "Spalten" die drei Akteure des Systems. In der vierten "Spalte" ist das zu erstellende Software-System selbst aufgeführt und alle Pfeile, die dort beginnen oder enden, sind letztendlich Anforderungen, die von den Akteuren an dieses System gestellt werden. Das Zweck- und Tragweiten-Diagramm kann also aus diesem Sequenz-Diagramm abgeleitet werden.

Bei größeren Systemen werden Sie aber gar keine Chance haben, alle Prozesse verknüpft in einem einzelnen Diagramm unterzubringen. Dies gelingt schon allein deshalb nicht, weil an jedem Prozess viele Abteilungen und damit viele Abteilungsleiter und sehr viele Mitarbeiter beteiligt sind, die alle eine andere Sicht auf die jeweiligen Prozesse haben. Wenn es dann noch Zentralabteilungen oder Stabsstellen gibt, die "von außen" auf die Prozesse schauen, wird es völlig kompliziert.

Wenn das zu erstellende Software-System (die innere Systemgrenze im Zweck- und Tragweiten-Diagramm) im Laufe der Analysephase von der Blackbox zur Whitebox wird, dann werden die Prozesse, an denen das System beteiligt ist, deutlich sichtbar. Dann wird auch erkennbar, welche Prozesse nur teilweise von dem neuen System begleitet werden.

Es ist auf jeden Fall extrem wichtig, in der Pre-Analyse nur die **Fachlichkeit** des Kunden, also das reine **Geschäftsmodell** zu betrachten. Die Benutzeroberfläche oder technische Aspekte der Software spielen hier überhaupt gar keine Rolle! Noch mal: in dieser frühen Phase darf nur betrachtet werden, **was** gefordert wird, nicht **wie** es realisiert werden kann - es wird schließlich nichts eingebaut, nur weil es eingebaut werden *kann*, oder? Jeder Kunde kennt mindestens 90% seines Geschäftsmodells, weil er es jeden Tag lebt! Also sollten Sie als Analytiker nicht raten, wie etwas laufen könnte, oder an allen Ecken sofort vermeintliche Verbesserungen vorschlagen. Führen Sie stattdessen Interviews und lernen Sie den Kunden kennen: er sagt Ihnen schon, was Sie tun müssen, wenn Sie ihn nur fragen!

In [B13] gibt es ein Kapitel "2.3.4. Der Umgang mit politischen Risiken". Dieses Kapitel besteht aus nur zwei Sätzen: "Dazu kann ich keine ernsthaften Ratschläge geben, da ich kein »Unternehmenspolitiker« bin. Ich raten Ihnen nachdrücklich ab, jemanden zu finden, der dies ist." Das stimmt!

## 4.7 Angebote und Verträge

"Wohin wir auch blicken, überall entwickeln sich die Chancen aus den Problemen."  
(Nelson A. Rockefeller)

Die Pre-Analyse sollte zum Festpreis realisiert werden. Dadurch wird bereits ganz am Anfang des Projekts der wichtigste Aspekt eines jeden Projekts gepflegt: die Verbindlichkeit. Dem Kunden wird dabei klar, dass er sich auch für Software-Leistungen bewusst entscheiden und dafür auch bezahlen muss (was erstaunlicherweise vielen gar nicht klar zu sein scheint, weil sie offensichtlich immer noch viele fertige Software-Produkte kostenlos zu bekommen scheinen!) und der Anbieter wird gezwungen, ein Dokument (also etwas schriftliches) abzugeben. Gleichzeitig kann dabei gegenseitig das Verhalten bei der Geschäftsabwicklung (ich meine die Prozesskette "Angebot, Auftrag, Lieferung, Rechnung, Zahlung") erkundet werden.

"Murray wies immer darauf hin, wie wichtig es sei, einen Verkauf auch wirklich abzuschließen. Wir stellten fest, dass sich die meisten unserer Leute in den Anfangsstadien des Verkaufens gut bewährten, aber dann eine solche Angst vor Ablehnung hatten, dass sie potentielle Kunden oft wieder zur Tür

hinausgehen ließen. Sie brachten es einfach nicht fertig, zu sagen: *Unterschreiben Sie hier.*" ([\[B6\]](#) S. 57).

Der Aufwand für die Pre-Analyse kann bei zwei Stunden liegen, ich habe aber auch schon eine ganze Woche benötigt. Wichtig ist, dass hier auf jeden Fall zum Festpreis angeboten wird und das dieser Aufwand auch nicht überschritten wird, denn wenn die Pre-Analyse schon verbindlich gehandhabt werden kann, wie soll es dann bei der Realisierung aussehen? Achten Sie als Anbieter also darauf, dass sich bereits bei der Pre-Analyse Ihre ganze Professionalität zeigt!

Den Aufwand können Sie nur aus den Fakten des vorhergehenden Vertriebsgesprächs ermitteln. Falls es offensichtlich ein größeres Projekt wird (wobei natürlich jeder selbst entscheiden muss, was für ihn ein "größeres" Projekt ist), kann es Sinn machen, einen weiteren Kundentermin zu vereinbaren, bei dem Vertrieb und Analytiker zugegen sind. Dies ist insbesondere sinnvoll, wenn der Vertrieb nicht wirklich Ahnung von Softwareentwicklung hat und der Analytiker kein guter Vertriebler ist (und seien wir ehrlich: meistens ist doch beides der Fall, oder?).

Bedenken Sie bitte in jedem Fall, dass die Pre-Analyse per Definition unvollständig ist (s. "[Was also ist die Pre-Analyse?](#)")! Sie brauchen also nicht das Festpreis-Angebot zu scheuen.

Nach der Pre-Analyse sollte auch der Kunde ein Gefühl dafür bekommen haben, worauf er sich bei dem Projekt wirklich einlässt. Dieser Eindruck weicht häufig sehr stark von den ursprünglichen Vorstellungen ab. Da noch immer die meisten Unternehmen und Menschen recht wenig Erfahrung mit der Softwareentwicklung haben, sorgt die Pre-Analyse in der Regel für eine unangenehme Überraschung in Punkto Aufwand und Kosten. Oft wird hier dann schon zwischen Pflicht und Kür getrennt und eine Folgeversion geplant.

Je nach Projektgröße und Unternehmen wird entweder ein Festpreis-Angebot für die Analyse folgen oder ein Rahmenvertrag für den gesamten Rest des Projekts gemacht. Letzteres ist bei Festpreisprojekten immer schwierig, weil der Aufwand erst nach der Analyse genau genug bekannt. Bei einem reinen Aufwandsprojekt besteht für den Anbieter hier natürlich kein Risiko. Eigentlich tun sich beide einen Gefallen, wenn die Analyse zum Festpreis erfolgt und danach die einzelnen Anwendungsfallpakete ebenfalls zum Festpreis angeboten werden können.

## 5 Analyse

### 5.1 Analyseaspekte

**"Wenn wir nur zuerst herausfinden könnten, wo wir stehen und wohin wir tendieren, dann könnten wir besser beurteilen, was zu tun ist und wie es zu tun ist."**

(Abraham Lincoln)

Der Anbieter wird in dieser Phase zum Analytiker, der das Fachkonzept des Kunden strukturiert aufbereitet und in vielen Interviews ganz viele Fragen stellt, um die Antworten systematisch und strukturiert in die Analyse einzuarbeiten. Der Analytiker prüft alle Informationen auf Auslassungen, Widersprüche, Mehrdeutigkeiten, Ungenauigkeiten, Prioritäten und Irrelevanz. Stellt er Mängel fest, muss er Informationen nachfordern und nachbessern. Die Analyse etätigkeit verlangt also extrem viel Disziplin.

Es geht zunächst noch immer ausschließlich um die Fachlichkeit - also das Geschäftsmodell - des Kunden: alles, was die Software "wissen" muss, um die reale Welt des Kunden nachzubilden, gehört in das Analysedokument - aber mehr nicht! Das System soll ja auch nicht mit überflüssigen Funktionen überlastet und dadurch unbenutzbar werden! Es ist häufig extrem schwer zu beurteilen, was rein gehört und was nicht, d.h. zu bewerten ob es nötig ist, ein Detail zu spezifizieren, oder nicht. In [B4] gibt es phantastische Beispiele für softwareintensive Systeme, in denen Fehler teilweise mehrere Jahre gelauert haben, bis sich Mechanismen außerhalb des Systems oder implizite Voraussetzungen geändert haben, so dass die Fehler endlich zum Vorschein kommen konnte.

Ein Beispiel aus [B4]: "Der höheren Sicherheit wegen rüstete British Rail ihre Züge von Backenbremsen auf Scheibenbremsen um. Die neuen Bremsen arbeiteten gut, aber der Austausch verwirrte das Signalsystem gründlich. Das Signalsystem ist sicherheitskritisch, da sein Zweck die Verhütung von Zusammenstößen ist. Dabei greift es auf *drei* redundante computerisierte Systeme zurück, die die Position jedes Zugs über Sensoren in den Schienen ermitteln, die mit den Rädern des Zugs in Kontakt kommen. Die Position sämtlicher Züge ist im Stellwerk als Markierung auf einem Display verfügbar. Das Problem entstand im Herbst, als das Laub von den Blättern fiel und feuchtes Wetter herrschte. Nasse Blätter bildeten auf den Schienen eine breiige Masse und überzogen als isolierende Paste die Räder, die von den Scheibenbremsen nun nicht mehr wie von den Backenbremsen blank poliert wurden. Die Paste bildete eine immer dickere Schicht, bis die Züge keinen elektrischen Kontakt mehr zu den Sensoren im Gleis hatten. Aus Sicht des Signalsystems waren die Züge einfach verschwunden. Am Montag, den 11.11.1991 warteten Hunderte von Passagieren stundenlang auf ihre Züge, während die Betriebsleitung bei British Rail herauszufinden versuchte, welche Züge wo waren."

Dies belegt auch sehr schön, dass sich jedes Gesamtsystem aus Teilsystemen zusammen setzt, die sich gegenseitig beeinflussen. Selbst wenn die Entwickler des Signalsystems in ihrer Analyse der Anforderungen die Voraussetzung definiert hätten, dass die Räder immer Kontakt zu den Gleisen haben müssen, wäre dieser Fehler aufgetreten, weil die Verantwortlichen des Bremssystems vor dem Umbau der Bremsen bestimmt nicht die Spezifikation des Signalsystems gelesen hätten. Hier hatte eine unbeabsichtigte und unbewusste Funktion des alten Bremssystems (nämlich das sauber halten der Räder) zufällig zu einer unbedingten Voraussetzung der Funktionalität des Signalsystems geführt - und da hilft auch die Redundanz von *drei* Computern nicht weiter. Es gibt wohl auch kaum keine Möglichkeit, solche Zusammenhänge in die Spezifikation des Gesamtsystems mit aufzunehmen, weil diese zufälligen Zusammenhänge niemandem bewusst sind.

Sehr wohl zeigt sich gerade in solchen Fällen, ob professionell gearbeitet wird: "Professionalität" setzt sich hier aus strenger Formalität, viel Systematik, noch mehr Test auf Vollständigkeit und Widerspruchsfreiheit sowie das Einnehmen verschiedener Positionen zur Ausübung verschiedener Sichten zusammen. Hier sei schon mal der besondere Charakter von "Fehlern" vorweg genommen, das sie umso billiger sind, je eher man sie findet und beseitigt [B16].

Die Analysephase wird wie jede Phase und der gesamte Software-Erstellungsprozess iterativ inkrementell durchlaufen, d.h. die Analyse wächst in mehreren Schritten. Bei jedem Schritt werden alle Bestandteile genauer beschrieben. So werden zunächst alle Bestandteil konzeptionell betrachtet, indem

nur kurz erläutert wird, welche Anforderung überhaupt dazu gehört. Danach wird konzeptionell kurz beschrieben, worum es bei jedem Bestandteil geht. Als nächstes wird spezifiziert, wie der Bestandteil genau aussieht. Schließlich werden erstmalig Realisierungsaspekte ins Spiel gebracht. So wird aus einer sehr rohen Liste (Anforderungen) eine detaillierte Beschreibung (Spezifikation).

## 5.2 In/Out-Liste

Bei der Analyse ist wichtig, dass sehr viel Klärung stattfindet: alle Anforderungen müssen aus- und bewertet und gegen das [Hauptziel](#) getestet werden, das während der Pre-Analyse definiert wurde. Es müssen also **Entscheidungen** getroffen werden, ob eine an das System gestellte Anforderung wirklich realisiert werden soll oder nicht. Bei vielen Entscheidungen werden Sie als Analytiker feststellen, dass sich Ihr Kunde sehr schwer tut und schwankt und eine große Unsicherheit an den Tag legt. Es ist aber wichtig, dass eine Entscheidung getroffen wird, die dann auch genauso formal und verbindlich wie alles andere behandelt wird. Erstellen Sie eine In/Out-Liste und nehmen Sie die Entscheidung dort auf:

In	Out	Beschreibung	Datum
-	Out	Erfasste Artikel werden mit Zeitstempel versehen	2002-07-15
In	-	Stücklisten	2002-07-19

Dieses Instrument ist höchst einfach, aber genauso effektiv. Es entspricht einer meiner Grundregeln für jedes Projekt: **KISS (Keep it small and simple)**. Eine Entscheidung kann ja durchaus später revidiert werden, aber dann ist wenigstens klar, dass die Konsequenzen (d.h. der zeitliche und damit finanzielle Aufwand) ermittelt und in Rechnung gestellt werden können. Manchmal ist es auch sinnvoll, die In/Out-Liste parallel auf einem Flipchart-Blatt zu führen und dieses gut sichtbar im Projekt-raum aufzuhängen. Dann kann man immer schön mit dem Finger drauf zeigen :-)

Wenn bereits klar ist, dass die Software in mehreren Schritten erstellt wird, so kann der In/Out-Liste natürlich auch eine Spalte hinzugefügt werden, in der bei einem "In" eine Versions- oder Modulnummer oder vielleicht auch ein Datum eingetragen wird. Die Liste hilft in jedem Fall ganz deutlich dabei, die **Pflicht und Kür** zu trennen: die "Pflicht" muss rein und die "Kür" fliegt raus oder wird dann gemacht, wenn die Pflicht erledigt ist. Dies ist wieder eine meiner Grundregeln für jedes Projekt: Erst die Pflicht, dann die Kür!

### Protokolle

Dies ist eine gute Stelle, um ein leidiges Thema anzusprechen: Ich verstehe nicht, warum Projektteilnehmer so ungern Protokolle erstellen! Vielleicht liegt es daran, dass man sich festlegen muss. Vielleicht ist es auch, weil man dann zu einer Entscheidung stehen muss und später festgestellt werden kann, wer den Fehler gemacht hat. Es ist wahrscheinlich genau die große **Verbindlichkeit**, die ein Protokoll bedeutet, wovor sich die Menschen fürchten. Aber gerade diese Verbindlichkeit sorgt stets für klare Verhältnisse im Projekt und damit für ein hohes Maß an Objektivität und Fairness. Ich habe das Thema "Verbindlichkeit" bereits in der Pre-Analyse angesprochen und werde beim Design erneut darauf zurück kommen, denn es zieht sich wirklich durch das gesamte Projekt!

Eine Entscheidung, die zu einem "In" oder "Out" führt, wird in einer Sitzung mit dem Kunden getroffen. Zu einer Sitzung mit dem Kunden gehört ein Protokoll, das bei mir immer wie folgt aussieht und mit Microsoft Word geschrieben wird:



[großes Protokoll-Bild \(116 kb\)](#)

Sie sehen in dem Protokoll Unterschriftsfelder, denn ich drucke tatsächlich jedes Protokoll zwei mal aus; mein Ansprechpartner des Kunden und ich unterschreiben beide Protokolle, so dass jeder ein

Exemplar mit beiden **Unterschriften** hat und dieses in seinem Projektordner abheften kann. Dabei folge ich einfach dem Motto: "Was am Freitag nicht unterschrieben ist, ist am Montag vergessen." Spätestens bei der Unterschrift merkt der Kunde, dass ich es ernst meine und er überlegt sich seine Ins und Outs sehr gut. Ich habe allerdings noch nie ein Problem damit gehabt, die Unterschrift zu bekommen, denn auch der Kunde lernt diese Verbindlichkeit sehr schnell zu schätzen. Probieren Sie es mal aus!

Die Protokolle sind natürlich in der **Sprache** des Kunden zu führen. Ich erstelle Programmdokumentationen, Team-Spezifikationen und Quellcode-Kommentare grundsätzlich in Englisch, weil zum einen alle Programmiersprachen in Englisch arbeiten und damit auch englische Variablen-, Funktions- und Klassennamen selbstverständlich sind, und weil zum anderen zumindest bei größeren Projekten doch häufiger Leute beteiligt sind, die immer mindestens Englisch verstehen können. Alle Dokumente, die der Kommunikation mit dem Kunden dienen, müssen aber natürlich in der Sprache des Kunden erstellt werden.

### 5.3 Interviews

**"Das Problem zu erkennen ist wichtiger als die Lösung zu finden, denn die genaue Darstellung des Problems führt fast automatisch zur richtigen Lösung."**  
(Albert Einstein)

Es ist äußerst wichtig, dass es bei jedem Projekt sowohl auf Anbieter- als auch auf Kundenseite genau einen verantwortlichen Ansprechpartner - also einen Vertreter - gibt, denn in der Regel wird nur bei der beidseitigen Bündelung der Verantwortung die Verbindlichkeit akzeptiert. Das heißt aber nicht, dass der Vertreter des Kunden auch die gesamte Fachlichkeit des Kunden beherrschen muss. Gerade bei größeren Projekten kann eine Person alleine unmöglich die gesamte Fachlichkeit des Kunden beherrschen. Daher sollte der Vertreter des Kunden für jeden Akteur stellvertretend einen Vertreter suchen, bei dem er sich erkundigen kann. Die Verantwortung für das Ergebnis liegt gegenüber dem Anbieter dann natürlich wieder bei dem einen einzigen Vertreter des Kunden, aber die Sicherheit und Qualität der Antworten steigt und es vergeht dennoch nicht viel Zeit bis zur Antwort.

Es macht durchaus sehr viel Sinn, dass der Vertreter des Anbieters an diesen fachlichen Interviews der Vertreter der Akteure teilnimmt, zum einen um den Stille-Post-Effekt zu vermeiden, zum anderen aber auch, um sich selbst einen Eindruck von der Sicherheit in der Fachlichkeit der Akteure zu verschaffen. Es kommt schließlich nicht selten vor, dass auch die Akteure selbst die Fachlichkeit nicht sauber formulieren können, weil sie nie systematisch darüber nachgedacht haben, warum sie was wie tun.

Nicht selten kommt es vor, dass ein "Cheffe" meint, am besten zu wissen, was seine Leute wie tun, weil er es ihnen vorgibt. Meine Erfahrung hat aber gezeigt, dass dies noch lange nicht heißt, dass seine Mitarbeiter auch wirklich so arbeiten. Oft müssen die Mitarbeiter "Korrekturen" ansetzen, die sie auch nur zugeben können, wenn der Vorgesetzte bei den Interviews nicht teilnimmt. Wie bei allen anderen Menschen kann auch Chefs nur geholfen werden, wenn sie wirklich bereit sind, sich helfen zu lassen).

Ich denke, nach dem vorhergehenden Unterkapitel ist es selbstverständlich, dass bei jedem Interview ein Protokoll erstellt wird und dass der Ansprechpartner des Kunden ebenfalls an dem Interview teilnimmt und dass beide "Parteien" das Protokoll unterschreiben.

Ich nehme grundsätzlich in das Analysedokument eine kleine Tabelle mit auf, wann mit wem ein Interview stattgefunden hat. Ich vermerke in dieser Tabelle auch, ob die Ergebnisse des Interviews bereits in die Analyse eingearbeitet sind. Ich nehme grundsätzlich alle Ergebnisse (also insbesondere alle Anforderungen) in die Analyse mit auf, kennzeichne dann aber mit meinem Ansprechpartner zusammen die Anforderungen, die nicht als "offizielle" Anforderungen des Kunden stehen bleiben sollen (natürlich erstellt hier jeder Interview-Partner einen fetten Wunschzettel!).

Fachabteilung Gruppe	Datum	in Analyse eingearbeitet
Buchhaltung	2002-07-18	ja
Lager	2002-07-19	nein

## 5.4 Problemzerlegung

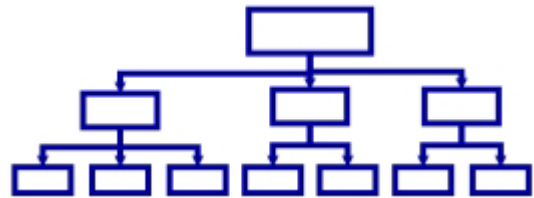
**"Verwandle große Schwierigkeiten in kleine und kleine in gar keine."**

(aus China)

Wie im Eingang dieses Kapitels erläutert, geht es bei der Analyse darum, das Hauptziel des Projekts über das Fachkonzept und die vielen geführten Interviews in Teilziele und Teilprobleme mit Teillösungen zu zerlegen und die ursprünglich gestellten Anforderungen systematisch einzuarbeiten. Im Eingang dieses Kapitels habe ich auch zur Sprache gebracht, dass die Analyse iterativ inkrementell erstellt wird. Dabei sind die im Folgenden erläuterten Methoden hilfreich (die übrigens alle auch schon vor der Objektorientierung in der Software-Entwicklung angewandt wurden). In den folgenden Kapiteln wird dann eine Arbeitsweise vorgestellt, die diese Methoden konsequent anwendet.

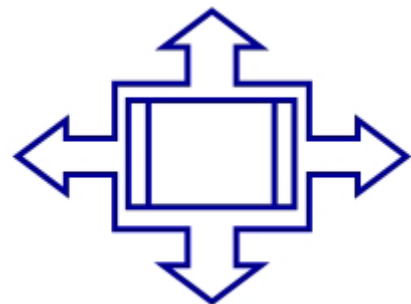
### 5.4.1.1 Hierarchisierung

Die Hierarchisierung dient der Zerlegung eines Problems in Teilprobleme. Die Teilprobleme können dann isoliert für sich betrachtet werden. Es ist immer einfacher mehrere Teilprobleme zu lösen als ein Gesamtproblem. Die Vorgehensweise ist hierbei typischerweise top-down (von oben nach unten), weil ein großes Problem in mehrere mittlere Probleme zerlegt wird, die dann in noch mehr kleine Probleme zersplittert werden. So wird das Projektziel in Teilziele und weiter in konkretere Schritte zerlegt, ganz nach dem chinesischen Sprichwort: "Verwandle große Schwierigkeiten in kleine und kleine in gar keine". Aus den Teilzielen ergeben sich bei der späteren Realisierung dann auch die Arbeitspakete und damit die Projektmeilensteine und Programmversionen. Weil ganze Sub-Bäume der Hierarchie voneinander getrennt verlaufen, können in den späteren Projektphasen auch mehrere Entwickler oder gar mehrere Entwicklerteams parallel arbeiten und auch die Qualitätssicherung und die Handbuchsreiber können ihre Arbeit besser aufteilen.



### 5.4.1.2 Modularisierung

Aus der Zerlegung bei der Hierarchisierung entstehen einzelne Module, die nur über definierte Schnittstellen mit den umgebenden Modulen kommunizieren können und ansonsten kontextunabhängig sind. Wenn diese Schnittstellen vorab präzise definiert werden kann ein Modul in späteren Projekten vielleicht auch wieder benutzt werden und zum treuen Begleiter über mehrere Jahre werden. Durch diese Wiederverwendbarkeit verringert sich in den Folgeprojekten der Entwicklungsaufwand und gleichzeitig erhöht sich die Robustheit der Software, weil bereits getestete, erprobte und sogar bewährte Module zum Einsatz kommen. Diese Module werden zunächst sicherlich in Prosa mit dem Kunden besprochen und dann iterativ immer präziser spezifiziert und durch Änderungsanforderungen im Laufe der Zeit auf dem aktuellen fachlichen Stand gehalten.



Bei der späteren Programmierung der Module werden Code und Daten der gekapselt und über eine gemeinsame Schnittstelle von der Umgebung isoliert. Daher können auch Fehler recht schnell auf

Module eingegrenzt und damit lokalisiert werden. Ändert sich durch die Fehlerbehebung die Schnittstelle des Moduls nicht, so ist in der Regel auch der Folgetest nach der Änderung auf eine kleine Anzahl von Modulen begrenzt. Diese Aspekte zeigen deutlich die Wichtigkeit sowohl von Definition als auch von Dokumentation der Schnittstellen.

### 5.4.1.3 Strukturierung

Die Module selbst ergeben zusammengesetzt wieder die Hierarchie, wobei die Schnittstellen das Zusammenspiel der Module definieren. Die Module selbst sind aus Sicht der Hierarchie Black Boxes, denn es sind nur deren Namen, Aufgabenbeschreibungen und Schnittstellen, aber nicht die Inhalte sichtbar.

Bei der fortschreitenden Entwicklung des Software-Systems müssen natürlich alle Module spezifiziert und letztendlich programmiert werden. Hierfür ist übrigens ein außerordentlich hohes Maß an Disziplin erforderlich, weil sehr viel Detailkram entdeckt, analysiert, diskutiert, hinterfragt, dokumentiert, getestet und manchmal auch wieder verworfen werden muss. Dabei das gesamte Analysedokument stets aktuell und konsistent zu halten, erfordert wirklich eine immer wieder erstaunliche Akribie.

Jeder gefundene Lösungsansatz einer analysierten Fragestellung wirft in aller Regel wieder weitere Fragen auf, die wieder analysiert werden müssen. Dies scheint lange Zeit kein Ende zu finden, weil der Fragenberg immer größer statt kleiner wird. Dabei ist es sehr wichtig, stets das passende Abstraktionsniveau zu finden und zu halten. Gegebenenfalls muss ein Modul nochmals gesplittet werden, weil sich bei der Strukturierung des Moduls noch zuviel Komplexität zeigt. Vielleicht zeigt sich aber auch, dass es in dem Modul kaum etwas zu strukturieren gibt und das Modul sehr gut mit einem Nachbarmodul verschmolzen werden kann. An dieser Stelle wächst oft die Ungeduld, weil man sich schon viel weiter wähnte, als man in Wirklichkeit ist. Aber unser Kopf ist schließlich rund, damit das Denken die Richtung ändern kann - also sollte man die neu gewonnenen Ansichten auch umsetzen!

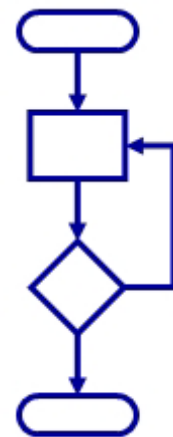
Miller [L8] zeigte schon sehr früh, dass der Mensch sehr gut  $7 \pm 2$  Aspekte überschauen kann. Daran sollten Sie sich orientieren, wenn Sie Ihr Problem zerlegen, hierarchisieren und aufgrund der Ergebnisse der Strukturierungsansätze die Module neu verteilen. Nach jeder Neuverteilung können Sie die nächste Iteration einläuten, also wieder auf Lücken und Widersprüche prüfen und die angepassten Module neu strukturieren.

Für viele Programmierer hat dies nichts mit Programmierung zu tun und sie haben Recht! Denn dies hat vielmehr was mit Softwareentwicklung zu tun; die Programmierung ist nur ein Teil der Softwareentwicklung, aber eben genau der, den die meisten Softwareentwickler am liebsten mögen. Daher wählen sie oft eine vermeindliche Abkürzung und lassen den ganzen Analyse-, Spezifikations- und Dokumentationskram einfach weg. Der Erfolg gibt ihnen zunächst Recht, denn sie haben sehr schnell ein Stück Software am Laufen - aber die Rache der Software ist stets so groß, dass die gewonnene Zeit schnell wieder aufgebraucht ist - von Stabilität, Wartbarkeit und Wiederverwendbarkeit ganz zu Schweigen!

In der Regel wird bei der Programmierung der fertig spezifizierten Module die mittlerweile klassische Form der Realisierung mit Funktionen, Variablen, Entscheidungen, Schleifen, usw. eingesetzt. Auch bei der objektorientierten Programmierung sehen beispielsweise die Methoden der Klassen der Geschäftslogik oft sehr strukturiert aus! Es sollten allerdings stets möglichst moderne Verfahren für Fehler-Reporting, Transaktionskontrolle und Protokollierung benutzt werden. Der Entwicklungsrahmen muss natürlich auch Verteilungskonzepte für Server-Landschaften und Skalierbarkeit berücksichtigen.

## 5.5 Analysemuster (engl. analysis patterns)

...



## 5.6 Anwendungsfälle

"Es ist nicht genug zu wissen; man muss es auch anwenden.

Es ist nicht genug zu wollen, man muss es auch tun."

(Johann Wolfgang von Goethe)

Leistungsbeschreibung, Szenarien

einzelne Leistungsmerkmale

Anwendungsfälle ([Use Cases](#))

casual, fully dressed, Sequenzdiagramm, Prototyp ([s.u.](#))

Merkmal-Schnittstellen

[Alistair Cockburn, Humans and Technology](#)

Im Buch "[Writing Effective Use Cases](#)" zitiert Alistair Cockburn sehr passend ein kleines Gedicht des Iren Jonathan Swift:

*So, naturalists observe, a flea*

*Hath smaller fleas that on him prey*

*And these have smaller still to bite 'em*

*And so proceed ad infinitum.*

Ich betone hier deutlich, dass ich aus meiner Erfahrung heraus NICHT mit Cockburn darin übereinstimme, dass es zwei Stile für die Beschreibung von Anwendungsfällen (nämlich casual und fully dressed) gibt und man sich im Projekt für einen der beiden Stile entscheiden wird (S. 9). Vielmehr habe ich festgestellt, dass es Sinn macht, für jede Anwendungsfallbeschreibung einzeln zu entscheiden, welche Form man wählt. Ich halte es manchmal für sehr sinnvoll, einen Anwendungsfall als Sequenzdiagramm oder als Screenshot aus dem Prototypen ([s.u.](#)) zu beschreiben. Das sagt manchmal viel mehr aus als jede fully dressed Beschreibung.

Akzeptanzkriterien für Anwendungsfälle (mindestens eins) ([Rösch Consulting](#))

s. auch "[Systematische Tests](#)"

z.B. konkretes Beispiel

konkrete Ziele definieren

abnahmefähig (juristisch stabil)

fachlich fehlerfrei! Kundenverantwortung

Anwendungsfallfragmente zur Wiederverwendung von Fachlichkeiten

Aspektororientierte Programmierung (AOP)

Mit den Anwendungsfällen sollten sofort auch die [Testfälle](#) erstellt werden.

## 5.7 Spezifikationsmuster

Wenn während der Analyse der Anforderungen die konzeptionelle Sicht auf die Fachlichkeit erledigt ist und die Anwendungsfälle alle im Rohbau stehen, wird es Zeit, sich über Prototypen oder Prosatexte für die genauere Spezifikation als Vorgabe für die Implementierung der **Programmpäsentation** Gedanken zu machen. Hier kommt es zu solchen Diskussionen zwischen Analytiker und Kunde: "Wie wird denn bei Ihnen eigentlich eine Stückliste erstellt?" "Wir nehmen ein neues Stücklistenformular, tragen eine neue Nummer ein und ordnen dieser Liste verschiedene Artikel zu." "Und wie viele Artikel kann eine Stückliste umfassen?" "Oh, das ist ganz verschieden - aber mindestens zwei, sonst ist es keine Stückliste." "Und wie erfolgt die Zuordnung?" "Wir tragen einfach die Nummern der Artikel in das Formular ein."

Dieses Beispiel zeigt die Verwendung typischer **Begriffe**: Liste, erstellen, Formular, eintragen, zuordnen. Solche Worte kommen immer wieder vor und führen in der Regel auch zu einer ähnlichen Implementierung. Statt einer langen detaillierten Prosabeschreibung oder einem Prototypen für die Bildschirmdarstellung könnte man im obigen Beispiel auch einfach schreiben: "Die Nummer der Stückliste wird eingegeben und die Artikel werden zugeordnet." Wenn Analyst und Kunde einmal definiert haben, was "eingeben" und "zuordnen" bedeutet, ist dieser eine Satz schon eine sehr genaue Spezifikation.

Immer wieder vorkommende Elemente, Begriffe oder Konstellationen werden auch als "Muster" bezeichnet. In diesem Fall dienen die erkannten Muster zur Vereinfachung der Spezifikation, daher bezeichne ich sie als "**Spezifikationsmuster**" (englisch *Specification Patterns*). Ich muss allerdings betonen, dass es sich hierbei jetzt nicht um ein allgemein bereits anerkanntes oder gar genormtes Instrument handelt. Betrachten Sie diese Art der Muster als Vorschlag meinerseits. Ich verwende sie derzeit in einem recht großen Projekt und alle Beteiligten kommen sehr gut damit klar.

Ich unterscheide zwischen Spezifikationsmustern zur "Ablaufsteuerung", mit denen der Bedienfluss innerhalb der Software gesteuert wird und Mustern zur "Inhaltssteuerung", mit denen die Erfassung und Darstellung von Informationen gesteuert wird. Da die Visualisierung der Muster bei einem Windows-Programm anders ist als bei einer Internetseite oder auf einem WAP-Handy, versuche ich, die optische Darstellung so **neutral** wie möglich zu beschreiben und eine Form zu wählen, für die es bei jeder erdenklichen Oberfläche eine Implementierungsmöglichkeit gibt. Auf jeden Fall sind die Spezifikationsmuster von der Fachlichkeit des Kunden unabhängig, so dass sie für alle neuen Kunden, Branchen und Projekte wieder verwendet werden können.

Die Spezifikationsmuster sind teilweise sehr einfach und atomar, teilweise aber auch sehr komplex und setzen sich aus weiteren Spezifikationsmustern zusammen oder verwenden weitere Muster in variabler Anzahl und Kombination.

#### Spezifikationsmuster zur Ablaufsteuerung

Ein **Assistent** führt den Anwender schrittweise durch einen programmierten Ablauf und sammelt auf seinem Weg Informationen ein, nimmt Einstellungen vor oder führt abhängige Arbeiten aus. Ein Assistent besteht typischerweise aus einer oder mehreren "Flächen". Das Spezifikationsmuster "Assistent" kann für das Analysemuster "Arbeitsprozess" angewandt werden. [Beispiele](#)

Eine **Fläche** dient zur Aufnahme weiterer Spezifikationsmuster und führt damit einerseits zur Kapselung dieser Muster und andererseits zur Isolation dieser Muster gegen andere. Dadurch wird die Wiederverwendbarkeit von bestimmten Muster-Konstellationen vereinfacht. Ein Fenster (bei einer Windows-Anwendung) oder eine Seite (bei einer Internet-Anwendung) besteht in der Regel aus einer Fläche und einem zusätzlichen Satz von "Aktionen" wie beispielsweise *OK* und *Abbrechen* oder *vor* und *zurück*.

Die **Datenerfassung** gruppiert mehrere Spezifikationsmuster auf einer "Fläche" zur Erfassung eines bestimmten Satzes von Daten. Dabei kann es sich beispielsweise um ein automatisch generiertes Formular zur Pflege einer Datenbanktabelle handeln. Auf einer Internetseite wird dieses Muster typischerweise auch als *form* implementiert. [Beispiele](#)

Eine **Aktion** wird immer auf die gleiche Art ausgeführt und präsentiert sich dem Anwender als feste Programmfunktion. Dieses Muster wird typischerweise als Schaltfläche implementiert, ein Steuerelement, das es bei nahezu allen Entwicklungsumgebungen gibt. [Beispiele](#)

An vielen Stellen soll der Anwender die Möglichkeit haben, zu einer Information eine **Detaillierung** durchzuführen, also weitere Details abzurufen. Es handelt sich hier eigentlich um eine "Aktion", aber um eine sehr gebräuchliche, die noch dazu fest mit dem Inhalt gekoppelt ist. Bei vielen Oberflächen ist es möglich, die zu detaillierende Information selbst besonders hervorzuheben und mit einer speziellen "Aktion" fest zu *verdrahten*. Ein gutes Beispiel ist der *Hyperlink* im Internet: er zeigt immer auf die Seite, die den Begriff, der als Link dargestellt ist, detailliert. [Beispiele](#)

Ein **Menü** bietet immer eine Auswahl an Programmfunktionalität und unterscheidet sich dadurch deutlich von der "Liste", die ausschließlich Werte (Daten) enthält. Das "Menü" wird in der Regel mehrstufig aufgebaut sein und sich inhaltlich von Stufe zu Stufe verfeinern. In jedem Softwaresystem sollte nur ein einziger Menübaum existieren. [Beispiele](#)

Die **Kaskade** ist von Wasserfällen her bekannt, wo das Wasser in mehreren Stufen bis auf das tiefste Niveau fällt. Es gibt in Softwaresystemen häufig Inhaltsbereiche, die sich stufig innerhalb einer "Fläche" ergeben. Oft sind zwischen diesen einzelnen Stufen "Aktionen" des Anwenders erforderlich. Ein "Menü"baum kann auch als "Kaskade" betrachtet werden; das passt allerdings nicht ganz, weil ein "Menü" ausschließlich Funktionalitäten des Programms enthält, wogegen eine "Kaskade" sich vorrangig mit Daten beschäftigt. [Beispiele](#)

#### Spezifikationsmuster zur Inhaltssteuerung

Fester, statischer **Text** auf einer "Fläche" dient der reinen Ausgabe einer Beschreibung, Bezeichnung oder Information. Der "Text" kann nicht durch den Anwender geändert werden kann. [Beispiele](#)

Eine **Eingabe** dient dem Anwender zum spezifizieren des Datenteils einer Information. Im Gegensatz zum "Text" kann die "Eingabe" vom Anwender geändert werden. Eine "Eingabe" kann auch leer sein, das Programm muss darauf entsprechend reagieren. Wenn ein Anwender nicht das Recht zum ändern hat, so sollte die "Eingabe" deutlich sichtbar gesperrt werden oder besser gleich stattdessen als "Text" angezeigt werden. [Beispiele](#)

Die **Voreinstellung** (englisch *default*) dient der Software als fest eingestellter Vorschlag für eine bestimmte noch leere "Eingabe" bei deren erster Verwendung. Es handelt sich also um einen in der Regel vom Kunden festgelegten Startwert. [Beispiele](#)

Oft soll der Anwender die Möglichkeit der **Wahl** eines Wertes aus einer Menge von mehreren Werten haben. Dabei wird diese Menge möglichst nur kurz angezeigt und der gewählte Wert bleibt stehen. Eine typische Implementierung mit VisualBasic ist die *ComboBox*, mit HTML der *Select*. Eine Erweiterung ist die "Wahleingabe" und eine besondere Rolle spielt die "Ja/Nein-Wahl" sowie die "Optionen-Wahl". [Beispiele](#)

Eine **Liste** enthält eine lineare Anordnung (Aufzählung) von "Detaillierungen". Jeder einzelne Punkt dieser "Liste" ist also ein Spezifikationsmuster für die Ablaufsteuerung, aber die "Liste" selbst dient als Ganzes der Darstellung von Inhalt. [Beispiele](#)

Die **Festlegung** ähnelt sehr stark der "Liste", aber zusätzlich gibt es zu jeder "Detaillierung" noch die Zusatzaussage, ob das Element ausgewählt ist für Irgendwas oder nicht. Damit ergänzt die "Festlegung" die "Liste" um eine Art Filter. In der Regel wird hier festgelegt, welche Elemente für bestimmte "Aktionen" benutzt werden sollen und welche nicht. Bevor der Anwender die Elemente festlegt, kann er sich anhand der "Detaillierung" noch über die einzelnen Elemente informieren. [Beispiele](#)

Eine **Tabelle** ist eine Matrixanordnung von Daten, also so etwas wie eine zweidimensionale "Liste", also eine Übersicht von einem Wertetyp über einem anderen. Ein Teil der "Tabelle" kann auch wie eine "Liste", als einzelne "Detaillierung", als "Eingabe" oder einfach als "Text" ausgeprägt sein. [Beispiele](#)

Die **Zuordnung** ist quasi eine "Wahl" *einiger* Elemente aus vielen (die "Wahl" selbst kann nur genau *eins* aus vielen wählen). Der Anwender hat nicht wie bei der "Liste" die Möglichkeit, sich über die einzelnen Elemente zu informieren, weil diese nicht als "Detaillierung" ausgeprägt sind. Dafür kann der Anwender aber aus einem Vorrat beliebig viele Elemente einzeln wählen und einzeln abwählen. Ideal ist es, wenn der Anwender dabei eine Gegenüberstellung der bereits gewählten (links) und der noch wählbaren (rechts) Elemente sieht. [Beispiele](#)

Die **Einstellung** ist deutlich mehr als die "Eingabe". Sie wird immer dann genutzt, wenn ein Wert nicht einfach so eingetippt werden kann, sondern erst eine Ermittlung des Wertes erfolgen muss. Daher bietet die "Einstellung" eine separate "Fläche" zur Spezifikation des Wertes über die "Suche" oder prüft vor Akzeptanz des Wertes erst noch Plausibilitäten oder führt noch komplexe Zusatzaktionen aus. Die "Einstellung" eines Wertes wird häufig auch einmalig für mehrere "Aktionen" gleichzeitig durchgeführt, um den Dialog mit der Software zu vereinfachen. [Beispiele](#)

Anhand der **Suche** können Werte recht aufwändig ermittelt werden. Hier können mehrere andere Spezifikationsmuster kombiniert werden, um dem Anwender einen gewissen Komfort zu bieten. Die "Suche" kann auch bei der "Einstellung" mitbenutzt werden. [Beispiele](#)

Weitere Gedanken

Sie erhalten durch diese Muster die Möglichkeit, mit einem einzigen Prosasatz den **Prototyp** für eine ganze Funktion darzustellen. Es könnten Ihnen als Analytiker also passieren, dass Sie irgendwann einen Anruf vom Kunden erhalten und folgenden Satz hören: "Können wir den *Assistenten* von Anwendungsfall 85 noch so ergänzen, dass er auf der Seite 1 eine absteigende *Liste* von Jahreszahlen mit *Detaillierungen* zum Wechseln auf Seite 2 und eine *Aktion* "Ändern" zum Wechseln auf eine neue Seite 4 enthält?" Beide haben dann automatisch das gleiche Bild vor Augen und können präzise Funktionalität und Aufwand aushandeln - und zwar am Telefon!

Spezifikationsmuster helfen beim Erlangen der **Implementierungssicherheit** für das zu erstellende Softwaresystem, denn die Implementierbarkeit eines solchen Musters ist sicherlich auf allen gängigen

Systemen gegeben. Wenn sich also ein Anwendungsfall weitgehend anhand der Spezifikationsmuster beschreiben lässt, so steht der Implementierung eigentlich nichts mehr im Wege.

Im ersten Moment könnte man denken, dass Spezifikationsmuster sich nur auf den Presentation Tier beziehen, weil sie die Interaktion zwischen Anwender und Software beschreiben und stets sehr gut helfen, sich einfacher ein Bild zu verschaffen als durch den Bau eines Prototyps. Aber bei vielen dieser Muster sind **Business Tier und Data Tier** sehr wohl beteiligt: wenn beispielsweise im Gewächshaus die Messwerte mehrerer Fühler in mehreren Kaskaden mit Zusatzinfos erfasst werden, so muss bei jeder Kaskade eine Interaktion mit dem Business Tier stattfinden, um den Messwert des nächsten Fühlers zu holen, und nachdem der Wert mit Zusatzinfos versehen ist, muss er vom Business Tier nochmals validiert und an den Data Tier zur permanenten Speicherung weiter gereicht werden.

## 5.8 Prototyp

**"Wenn dir das Leben Zitronen serviert mach einfach Limonade draus!"**  
(Unbekannt)

Es gibt jedoch Fälle, wo die Spezifikationsmuster allein nicht reichen. Dies kann daran liegen, dass die Situation einfach zu komplex ist, aber auch daran, dass ein Linienvorgesetzter, der das Vokabular und die Methodik natürlich nicht beherrschen kann, eine teure Entscheidung fällen muss. Oft sollen aber auch die gewählten Vertreter der Akteure sich vorab ein Bild von ihrer zukünftigen Software machen können, um noch Einwände oder Anregungen äußern zu können. Dann ist es erforderlich, einen möglichst realistischen Prototypen auf den Bildschirm zu zaubern.

Dafür kann schon die Erstellung eines statischen Screenshot-Prototyps mit Microsoft PowerPoint oder Microsoft Visio ausreichen. Dies hat dann auch den Vorteil, dass man auf evtl. vorhandenen "Photos" von ähnlichen Bildschirmen zurückgreifen kann, die man nur noch ergänzen muss.

Dynamische ("funktionierende") Prototypen können natürlich mit einem einfachen System statt der geplanten Echtumgebung (beispielsweise Access statt VB/SQL-Server) realisiert werden.

Es kann aber auch aus technischen Gründen erforderlich sein, einen - oder sogar mehrere - Prototypen zu bauen. Wenn der Anbieter beispielsweise noch keine Projekte auf der geforderten Plattform realisiert hat oder eine neue Technologie zum Einsatz kommen soll oder Zweifel an der Realisierung der Performance bestehen, können die Architekten, Designer und Entwickler ohne Rücksicht auf Programmierkonventionen oder Dokumentation Prototypen zusammenschießen, um zu prüfen, ob die aufgestellten Gedanken richtig sind.

Ein Prototyp ist kein Umweg! Zwar weiß man nach einem Projekt nicht, wie viel Zeit man durch einen Prototypen insgesamt gespart hat, aber der Vergleich zwischen ähnlichen Projekten mit und ohne Prototyp hat gezeigt, dass sich der Aufwand für den Auftraggeber bezahlt macht! Einige meiner eigenen Projekte wären ohne Prototyp gnadenlos gescheitert, weil nach der Diskussion des Prototyps die Gestaltung der Software massiv geändert wurde. Einmal wurde sogar wegen Uneinigkeit der Beteiligten über den Prototyp das ganze Projekt verworfen! Wie wäre das Projekt wohl abgelaufen, wenn es diesen Prototyp nicht gegeben hätte? Dies zeigt, dass man als Anbieter den Prototyp nicht nur unter dem Aspekt der gewonnenen Zeit verkaufen sollte.

Wie bereits beim Prototypen/Casey-Modell im Kapitel [Systematik](#) beschrieben, wird der Prototyp weggeworfen, schließlich ist es nur ein Prototyp: "Plan to throw one away; you will, anyhow." ("Planen Sie, einen wegzuworfen; sie werden es sowieso.") [\[B10\]](#). Hier sind sich nun wirklich alle einig (auch [\[B4\]](#) und [\[B12\]](#) beziehen sich auf [\[B10\]](#)). Sie sollten wirklich niemals den Prototyp in ein Echt-system wandeln, denn Sie haben bei der Entwicklung des Prototyps per Definition die vereinbarten Konventionen nicht eingehalten, keine Fehlerbehandlung eingebaut, nicht systematisch getestet und keine Dokumentation erstellt!

## 5.9 Testfälle

**"Die meiste Zeit geht dadurch verloren, dass man nicht zu Ende denkt."**  
(Alfred Herrhausen)

separates Dokument  
 priorisiert  
 Zeitdauer stoppen  
 aus Anwendersicht  
 mit den Anwendungsfällen zusammen erstellen  
 kann teilweise sogar der Kunde (Anwender) erstellen oder zumindest daran mitarbeiten

## 5.10 Dokumente

Pre-Analyse/Analyse/Spezifikation

Protokolle

Projektplan

Einsatzplan

Releaseplan

Risikoanalyse

"We usually foresee the future by reviewing the past, seeking longterm trends."

(dt.: "Wir sehen die Zukunft im allgemeinen vorher, indem wir die Vergangenheit prüfen und nach lang anhaltenden Trends suchen.")

("Deep Time" by Gregory Benford)

Kunde vergibt Prioritäten (hoch, mittel, niedrig) für Anwendungsfälle nach Wichtigkeit

Analytiker legt Risikograd für Anwendungsfälle fest

Anwendungsfälle auf Iterationen verteilen (höchstes Risiko zuerst!)

Aufwand für Anwendungsfälle schätzen (Mitarbeiter einbeziehen, Verantwortung delegieren, Expertenbefragungen, mehrere Schätzungen)

Manpower kalkulieren

+20% für Übergabe, +20% Buffer

Iterationsintervalle als Meilensteine definieren (MS Project)

Projektmanagement selbst als Aufwand berücksichtigen

Jedes Projekt entwickelt auch eigene Fachlichkeiten und damit ein eigenes Vokabular. Modern sind dabei die **TLA** geworden: **Three Letter Acronyms** (dreibuchstabile Abkürzungen). Ironischerweise ist TLA selbst ein Beispiel für eine TLA. Abkürzungen, aber auch Fachbegriffe des Kunden sowie technische Begriffe aus der IT sollten in einem **Glossar** geführt werden, in dem jeder Projektteilnehmer, jeder Pilotanwender oder einfach jeder Interessierte jederzeit nachschlagen kann. Glossar nur für Fachwörter des Auftraggebers und -nehmers, nicht für Informationen, Daten oder Algorithmen des Softwaresystems; diese gehören in die Spezifikation selbst.

The future IS uncertain!

The spec IS incomplete!

80% IS enough!

20% buffer time IS minimum!

"The Answer to the Great Question ... of Life, the Universe and Everything ... is ... Forty-two."

("The Hitchhiker's Guide to the Galaxy" by Douglas Adams, Chapter 27 - dt. "Die Antwort auf die Große Frage ... nach dem Leben, dem Universum und allem ... lautet ... Zweiundvierzig." aus "[Per Anhalter durch die Galaxis](#)" von Douglas Adams, Kapitel 27)

## 5.11 Gesamt-Anwendungsfalldiagramm

Beziehungen

enthält (includes, uses): Link

Generalisierung (generalization): Alternativen

erweitert (extends): Ableitung

Basis-Anwendungsfälle

Varianten-Anwendungsfälle

Geschäftsanwendungsfälle (Kunde, Ereignis)  
Systemanwendungsfälle (Software)

## 6 Design

### 6.1 Realisierung

**"Wer steilen Berg erklimmt, hebt an mit ruhigem Schritt."**

(William Shakespeare)

Spätestens nach Abschluss der Analyse wird ein Resümee über Machbarkeit und Sinnhaftigkeit gezogen. Dies kann dazu führen, dass das gesamte Projekt wegen zu hoher Risiken verworfen wird, wegen einer zu langen Zeitachse portioniert wird oder wegen der Überschreitung finanzieller Schwellen in Vergleichsangebote oder Ausschreibungen läuft.

Ich selbst habe auch schon ein Projekt erlebt, bei dem in der Mitte der Analysephase ein Zwischenresümee ergeben hat, dass die Anforderungen keineswegs einzigartig sind und wahrscheinlich von Standardprodukten abgedeckt werden können. Hier bestand die Gefahr, dass für einen einzelnen Kunden individuell ein Standardprodukt in der n-ten Ausprägung entwickelt wird. Die Anforderungsanalyse wurde daher nach fünf Tagen abgebrochen und durch eine Marktanalyse ersetzt. Tatsächlich konnte der Kunde schon nur wenige Wochen ein Standardprodukt einführen, das obendrein auch noch preiswerter war.

Wenn das Projekt machbar ist und der Kunde die Durchführung nach wie vor für sinnvoll hält und der Auftrag zur Realisierung (Implementierung, Konstruktion) erteilt wird, sollte es ein Kick-Off-Meeting innerhalb des Entwicklungsteams und ein weiteres Kick-Off-Meeting mit dem Kunden geben. Auf diesen Veranstaltungen können Mitarbeiter vorgestellt, Regeln definiert, Rollen verteilt und Vorgehensweisen vereinheitlicht werden. Außerdem ist die Stimmung zu dieser Zeit stets noch sehr positiv, so dass sich auch die Linienvorgesetzten aus den verschiedenen Eskalationspfaden im Guten kennen lernen können.

### 6.2 Kick-Off-Meeting

Historie des Projekts

Vorstellung des Kunden

Namen (Namenskürzel), Positionen (Organigramm) und Tel.-Nummern des Kunden

Vorgehensweise im Projekt (Modelle, Methoden)

Eckdaten des Projekts (time, budget, quality)

Rollen (Aufgaben) der Teammitglieder (Alias, Namenskürzel)

Eskalationspfad / Verhaltensregeln (Offen!, Ordentlich und Korrekt) / Kommunikationsregeln (Fragen!)

Rechte und Pflichten (Umgang mit Fehlern und KnowHow, Arbeitszeiten, Räume, Geheimhaltung, Datenschutz)

Kleiderordnung

Raumordnung für den Projektraum

Gesetze (KISS, Code-Optimierung)

Verbindlichkeit (Protokolle)

Terminplanung mit den wichtigsten Eckdaten des Projekts, Projektplan mit MS-Project (Balkendiagramm)

Konventionen, Definitionen

English names and comments please!

Name-Prefixes: c for class, i for int, s for String, ...; no underscores!

Alle Literale (beispielsweise 5 oder "Hallo") werden benannt

Use only int, double, String!

Use In-Line-Comments to make the source readable!

Alias-Stamps (Namenskürzel yyyy-mm-dd)

Die Formatierung der Blockklammerung ist frei

Die Plausibilität der Parameter wird immer in(!) der Funktion geprüft  
 Eine Klasse pro Quelle, Quellenverwaltung (beispielsweise VSS oder CVS)  
 Fehlerbehandlung und Log-Mechanismus sind zu benutzen und ebenfalls zu testen  
 Jede Klasse wird vor Veröffentlichung getestet und abgenommen  
 Änderungskennzeichen für Dokumente (beispielsweise ## oder \*\*)  
 Bei Dokumenten Dateinamen "yyyy-mm-dd ..."  
 Projektkennwort (beispielsweise zur Verschlüsselung von eMails)  
 Projektleitungskreise und Projektsteuerungsgremien

## 6.3 Architekturen

### 6.3.1.1 MVC Paradigma

MVC steht für "**Model-View-Controller**" und geht auf das EVA-Prinzip zurück: die Eingabe, Verarbeitung und Ausgabe von Daten sollte immer getrennt betrachtet werden. Im Zusammenhang mit Smalltalk wurde in den 70er Jahren im Xerox PARC das EVA-Prinzip erstmalig auf die graphische Benutzerschnittstelle übertragen und als weiterentwickelte MVC-Architektur etabliert.

Das **Model** ist die genau eine Programm-Repräsentation, stellt also die eigentlichen Daten und Funktionalitäten zur Verfügung, die eng mit dem Problem verbunden sind und deshalb so lange unverändert bleiben, wie auch das Problem unverändert besteht. Die **Views** sind all die Präsentationen des Modells gegenüber dem Anwender, also auch die echte Visualisierung der Daten und damit die Verbindung des Modells zur Aussenwelt. Die **Controller** steuern das Verhalten der Software, nehmen also Ereignisse vom Anwender aus den Views entgegen, beeinflussen das Model und führen dadurch zur Veränderung der Views.

Strikt genommen kann das Model in ein Domain Model ( $M_D$ ) und ein Application Model ( $M_A$ ) geteilt werden: das  $M_D$  kümmert sich *ausschließlich* um das eigentliche fachliche Problem, wogegen das  $M_A$  die Schnittstellen zu den Views und Controllern enthält. Die Kommunikation zwischen  $M_D$ ,  $M_A$ , V und C erfolgt häufig über Ereignisse (englisch **events**) oder Nachrichten (englisch **messages**).

Die Begrifflichkeiten sind hier übrigens nicht immer sauber abzugrenzen: MVC wird je nach Kontext als Muster, Idiom, Paradigma, Prinzip, Architektur oder auch Modell betrachtet. Wenn Sie weitere Informationen zu MVC suchen, geben Sie einfach in einer Internet-Suchmaschine "MVC Smalltalk" ein. Sie werden beispielsweise [dieses](#) finden.

#### n-tier-Model

Bei jeder auch nur halbwegs ernst gemeinten Software für einen kommerziellen Kunden mit mehr als drei Anwendern sollte die Architektur immer eine Zerlegung in Komponenten mit einer Verteilung dieser Komponenten im Netzwerk ermöglichen. Der [CORBA-Standard \[L16\]](#) der [OMG \[L3\]](#) beschreibt seit Jahren, wie eine Software auf mehrere Computer verteilt werden kann. "**CORBA**" ist die Abkürzung für "Common ORB Architecture" und "ORB" ist die Abkürzung für "Object Request Broker". Ein ORB ist eine Systemsoftware, die im Netzwerk mit Objekten - also Softwaremodulen - makelt. Wenn also ein Objekt ein anderes im Netzwerk sucht, kann der ORB dieses finden. CORBA ist ein allgemeiner Architekturstandard für die Implementierung von ORBs in ein System. Bekannte CORBA-Implementierungen sind beispielsweise Microsoft-COM (Component Object Model) und EJB (Enterprise Java Beans).

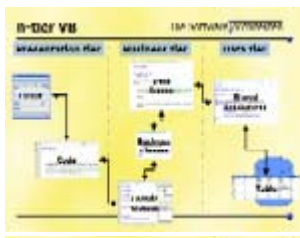
Es ist also sehr wohl möglich, von einem Programm auf dem einen Computer eine Funktion in einem Programm auf einem anderen Computer aufzurufen. Dies führt in der Praxis dazu, dass heutzutage die wesentliche Logik der Software von den Modulen auf den **zentralen Servern** erledigt wird, denn hier steht die größte Rechenleistung zur Verfügung und hier können die Änderungen so durchgeführt werden, dass sie sofort allen Anwendern zur Verfügung stehen. Die Software-Module, die wirklich auf den **Anwender-PCs** installiert werden, kümmern sich heutzutage nur noch um die reine Kommunikation mit dem Anwender, d.h. die Eingabe der Daten, das Auslösen von Aktionen und die Präsentation der Ergebnisse. Die Daten werden auf speziellen **Datenbank-Servern** gespeichert, wo sie gegen direkte Zugriffe durch die Anwender geschützt sind. Selbst die Software-Module auf den Anwender-PCs fordern diese Daten indirekt über die zentralen Logik-Server an.

Damit haben wir in der PC-Welt sowohl die Zeit hinter uns gelassen, in der die Software ausschließlich auf den PCs läuft, als auch die Zeit, in der nach **Client-/Server-Technik** die gesamte Logik auf

dem PC läuft und nur die Daten, Dateien und Drucker zentral verwaltet werden. Heute kann auch nicht mehr nur vom "3-tier-Model" gesprochen werden, weil größere Programme noch feiner modularisiert und im Netz verteilt werden.

Das n-tier-Model ermöglicht im Wesentlichen die **Kapselung** und **Trennung** von Wissen im Software-System. So kann beispielsweise die Umrechnung von Währungen oder die Ermittlung von Rabatten mit speziellen Modulen auf speziellen Servern realisiert werden. Das Wissen über die Umrechnungsfaktoren der Währungen oder der Staffeln des Rabattsystems liegt dann in zentralen Komponenten und kann in der Regel ohne Einfluss auf den Rest des Software-Systems geändert werden. Noch wichtiger ist anders herum, dass alle anderen Module des Software-Systems zwar wissen, **was** es gibt und dies auch benutzen können, aber nicht wissen müssen, **wie** es funktioniert und wie es implementiert ist.

Die zentrale Schicht, die als "Business Tier" die Geschäftslogik implementiert, weiß also nicht, ob oder gar wie die Daten in einer Datenbank gespeichert werden; sie weiß auch nicht, welche Daten dem Anwender wie präsentiert werden. Durch die Kapselung des Wissens in Module entsteht also gleichzeitig eine Trennung des Wissens und durch klar definierte **Schnittstellen** kann die Entwicklung der Software auch leicht auf mehrere Entwickler gleichzeitig verteilt und nahezu beliebig skaliert werden.



[große n-tier-Graphik VB \(354 kb\) mit weiteren Erläuterungen](#)



[große n-tier-Graphik Java \(408 kb\) mit weiteren Erläuterungen](#)

Beispiel

Von Sun™ gibt es für den Sun™ ONE (iPlanet™) Application Server eine Beispiel-Applikation "Java Pet Store", deren [Architektur](#) im Internet öffentlich beschrieben ist (alternativ [hier](#) als mirror).

## 6.4 Design der Modelle

"Design" bedeutet hier nicht das "Aussehen" der Software, sondern das **"Entwerfen"** der Software, d.h. die Ableitung der wesentlichen Modelle aus der Spezifikation. Daran sollte auf jeden Fall der Analytiker beteiligt sein, der in der Analysephase des Projekts die Spezifikation mit dem Ansprechpartner des Kunden gemeinsam aufgestellt hat, weil dieser sowohl die Fachlichkeit des Kunden kennengelernt hat als auch die entscheidenden Mitglieder des Projektteams (oder zumindest den Projektleiter) wahrscheinlich schon lange kennt. Außerdem wird der Analytiker auch genug Erfahrung und aktuelles IT-Know-How mitbringen, um die wesentlichen Befürchtungen und Probleme des Entwicklerteams zu kennen.

In dieser Design- oder **Entwurfsphase** werden vorrangig drei Hauptmodelle entwickelt: das Datenmodell, das Objektmodell der Geschäftslogik (engl. business logic, also middle tier) sowie das Präsentationsmodell. Wenn Sie die Architektur nach dem n-tier-Modell entwerfen, werden Sie bei größeren Anwendungen auch Objektmodelle im presentation tier und data tier bauen, wenn Sie auf die Serverfarmen mehrerer Unternehmen gleichzeitig zugreifen, werden Sie mit mehreren Datenmodellen arbeiten und wenn Sie verschiedene Typen von Clients (Browser, Windows, Handy) implementieren werden Sie mehrere Präsentationsmodelle benötigen. Bei den meisten Anwendungen kommen Sie aber sicherlich mit den drei Hauptmodellen aus.

Die Erstellung der Modelle ist jeweils eine Wissenschaft für sich - es gibt beispielsweise Entwickler, die sich ausschließlich auf den Entwurf von Datenmodellen **spezialisiert** haben. Andere wiederum haben über mehrere Projekte hinweg im Laufe von einigen Jahren viel

Erfahrung mit ganz bestimmten objektorientierten Plattformen gesammelt und sind von daher absolut fit im Entwurf von Klassenhierarchien.

Es geht mir in dieser Abhandlung insgesamt um die systematische Arbeitsweise bei der Entwicklung von Software. Daher ist es mir sehr wichtig zu betonen, **dass** diese Modelle erstellt werden sollten, aber ich will hier nicht im Detail erläutern, **wie** das geht. Dafür gibt es bereits reichlich Literatur und Tools. Ich gebe daher nur einen kurzen Überblick.

#### 6.4.1.1 Datenmodell

In den Anwendungsfällen werden Sie bereits **Basisdaten** erkannt haben, die eigentlich weitgehend konstant sind und an vielen Stellen in dem Software-System benutzt werden, z.B. Städtenamen, Verbandsmitglieder, LKW-Typen, Lagerhallenbezeichnungen. Sie werden auch schon für die Funktionsanwendungsfälle Datengruppierungen erkannt haben, die dort typisch sind und als **Stammdaten** für mehrere Detail- oder auch Funktionsanwendungsfälle gelten, z.B. Artikeldaten, Kundendaten, Projektdaten, Rechnungsdaten, Adressdaten, technische Datenblätter.

Dies sind Ihre ersten **Entitäten** (engl. entities), also Einzelinformationen und Informationseinheiten. Sie werden auch schon **Beziehungen** (engl. relations oder relationships) zwischen diesen Entitäten erkannt haben, z.B. Stücklisten bestehen aus Artikeln, Kunden erhalten mehrere Rechnungen, Datenblätter gehören zu Maschinen. Ein Datenmodell wird auch als ER-Model (Entity-Relationship-Model) bezeichnet. Sie nutzen für die Erstellung des Datenmodells ein Werkzeug, dass manchmal in die Entwicklungsumgebung integriert ist, manchmal aber besser auch separat erworben wird. Sie können damit die Strukturen (Spaltennamen und -typen) Ihrer Datentabellen festlegen, Standardwerte für Informationen vorgeben, Regeln für Informationen definieren (z.B. "zwischen 0 und 100" bei Prozentangaben) und auch die Beziehungen zwischen den Datentabellen mitsamt der Kardinalitäten definieren.

Die Erstellung des Datenmodells ist ein erster akribischer **Test** der Spezifikation durch Entwickler, die mit der Fachlichkeit noch nicht sehr vertraut sind. Daher sollte der Analytiker dabei sein und sehr genau darauf achten, wo seine Spezifikation wackelt und daher noch weiterer Analysebedarf besteht.

Das nebenstehende Bild zeigt einen ersten Entwurf eines Datenmodells, der eine Woche Zeit beansprucht hat. Sie sehen an einigen Lücken, dass dieses Modell noch nicht fertig ist, aber Sie sehen an den sauber getrennten Bereichen auch, dass die Analyse und Spezifikation erfolgreich verlaufen war und klar abgegrenzte Übersichtsanwendungsfälle ergeben hat.

Die Designphase wird - wie alle anderen Phasen und überhaupt das ganze Projekt auch - **iterativ inkrementell** durchlaufen, d.h. in einem ersten Design-Durchlauf wird zunächst mit einer konzeptionellen Sicht grob entworfen, in einem weiteren Durchlauf werden mit spezifizierender Sicht Details definiert und in einem anderen Durchlauf wird dann mit implementierender Sicht eine Umsetzung in das echte System erfolgen. Bei größeren Projekten können hier noch mehr Iterationen stattfinden, wobei das System immer weiter verfeinert wird und daher inkrementell wächst.



[großes Bild \(87 kb\)](#)

Während gerade das Bild eine konzeptionelle Sicht zeigte sehen Sie jetzt nebenstehend ein Bild der implementierenden Sicht. Denn es wird zunächst in der konzeptionellen Sicht nur ein logisches Datenmodell erstellt, welches nur allgemeine abstrakte Datentypen kennt (z.B. Nummer, Text). Erst bei der spezifizierenden Sicht werden die **Datenbanksysteme** festgelegt, so dass dann die physikalischen Modelle erstellt werden können. Diese enthalten dann wirklich schon die Datentypen, Regeln und Beziehungen des jeweiligen Datenbanksystems. Mit der implementierenden Sicht werden dann wirklich die echten Datenbanken aus den physikalischen Modellen realisiert, die dann auch mit Daten gefüllt werden können.



[große Bild \(79 kb\)](#)

Das soeben gezeigte logische Modell wurde in dem Projekt sowohl in Microsoft Access (als Zwischenlösung für die Datenübernahme aus dem

Altsystem) als auch in Oracle (für die endgültige Lösung) implementiert. Dazu wurden aus dem einen logischen Modell zwei physikalische Modelle abgeleitet (was von ordentlichen **Modellierungstools** unterstützt wird), aus denen dann die Datenbanken halbautomatisch erstellt werden konnten. Dazu generieren die Tools Skripte, die dann im Datenbanksystem gestartet werden können oder die Tools erstellen direkt die Datenbanken in den Datenbanksystemen.

In dem genannten Projekt wurde nebenstehende Access-Datenbank direkt aus dem Tool erstellt. Hier muss zwar noch aufgeräumt werden, aber die meiste Arbeit hat das Tool erledigt - und zwar gar nicht mal schlecht.

Das nebenstehende Bild zeigt einen Ausschnitt des gleichen Datenmodells ein halbes Jahr später. Es sind Tabellen hinzugekommen, Beziehungen haben sich geändert, Typen wurden genauer spezifiziert (NULL zugelassen oder nicht) und Bereiche getauscht, aber das logische Modelle wurde konsequent gepflegt und sieht daher jederzeit aus "wie neu".



[große Bild \(258 kb\)](#)

Bei dem Design des Datenmodells ist darauf zu achten, dass es vollständig normalisiert wird, d.h. alle folgenden Normalformen durchlaufen werden:

- 1. Normalform:** Die Feldinhalte müssen einfach - also atomar - sein. Dazu werden die Daten in Tabellen mit Zeilen und Spalten eingetragen.
- 2. Normalform:** Jeder Zeile wird ein Schlüsselfeld hinzugefügt, über das jede Zeile eindeutig identifiziert werden kann. Jedes Feld muß funktional vom Schlüsselfeld abhängig sein. Der Schlüssel kann sich aus mehreren Feldern zusammensetzen.
- 3. Normalform:** Es dürfen keine funktionalen Abhängigkeiten zwischen Nicht-Schlüssel-Feldern existieren. Die Tabelle darf somit keine transitiven Abhängigkeiten aufweisen. Dazu wird die Tabelle so lange in kleinere Tabellen mit Verweisen zerlegt, bis funktionale Abhängigkeiten nur noch zwischen den Tabellen bestehen und nicht mehr zwischen Nicht-Schlüssel-Feldern.
- Boyce-Codd Normalform:** Sie liegt zwischen der 3. und 4. Normalform und arbeitet mit der Existenz von Kandidatenschlüsseln. Eine Tabelle befindet sich in der Boyce-Codd Normalform, wenn sie sich für alle Felder, die alternative Kandidatenschlüssel sind, in der dritten Normalform befindet.
- 4. Normalform:** Eine Tabelle darf keine paarweise auftretenden, mehrwertigen Abhängigkeiten aufweisen, d.h. die 4. Normalform erfordert, dass die Tabelle keine unabhängigen zusammengesetzten Primärschlüssel enthält.
- 5. Normalform (Project-Join-Normalform):** Eine Tabelle enthält keine paarweisen zyklischen Abhängigkeiten des Primärschlüssels, die aus drei oder mehr Komponentenfeldern zusammengesetzt sind. Relationen in Fünfter Normalform lassen sich nicht weiter aufteilen.

Die Normalisierung der Daten entfernt die **Redundanzen**, so dass die Daten klar beschrieben werden, eindeutige Beziehungen zwischen Datenteilen bestehen, die Datenintegrität gewährleistet ist, ein schneller Zugriff auf die Daten möglich ist und nicht zuletzt Platz beim Speichern der Daten gespart wird.

Es handelt sich bei der Normalisierung um eine **bewährte Technologie**, die schon seit vielen Jahren erfolgreich von Programmierern eingesetzt wird. Wird beim Design des Datenmodells von vornherein mit ER-Diagrammen gearbeitet und jeder Datentabelle grundsätzlich ein internes Schlüsselfeld zugeordnet, so sind bereits wichtige Schritte auf dem Weg zur normalisierten Datenbank erledigt.

Bei der späteren Implementierung wird zu beachten sein, dass je nach Bedarf verschiedene Methoden für den Datenzugriff existieren: bei tabellarischen Darstellungen wird eher per Abfrage (View) zugegriffen, bei einzelnen "punktuellen" Informationen wird eher eine Gespeicherte Prozedur (Stored Procedure) aufgerufen und bei komplexen Ermittlungen mit Fallunterscheidungen über mehrere Tabellen hinweg wird evtl. stufig mehrfach vom mittleren tier aus zugegriffen.

...

Modellierung

- Klassenhierarchie; Integrationsmodell (OOA-Modell)
- Datensammlung; Datenmodell (ER-Modell)
- GUI; Simulationsmodell (fachlicher Prototyp)

Identifikation von Klassen, Assoziationen, Aggregationen, Generalisierung, Subsysteme  
Zustandsautomaten

Architekturentwurf

- Anbindung an Benutzeroberfläche
- Anbindung an Datenhaltung
- Verteilung auf vernetzte Computer

Implementierungsentwurf

- Verfeinerung des Architekturentwurfs
- Anpassung an Programmiersprache

## 6.5 Design Patterns

"Gang of four"

[Patterns Home Page](#)

[Patterns and Software Essential Concepts and Terminology](#)

## 6.6 Software Ergonomie

First principle: know the user

Meet the expectations

Look and feel of GUIs: Style guides

The Dos and Don'ts

Screen design

Join the senses

Keep it small and simple (Kiss)

Help your user ("RTFM")

## 7 Implementierung

### 7.1 Implementierungsaspekte

**"Amateure bauten die Arche, Profis bauten die Titanic."**

(Unbekannter)

Höchste Risiken zuerst erledigen

Aktivitätendiagramme

technische Entscheidungen

Jedes Softwareprodukt hat einen Helden, der einen Teil seines Lebens für die Fertigstellung der Software geopfert hat. In unendlich vielen Stunden hat er das Projekt fast von Anfang an bis fast zum bitteren Ende begleitet und stellt die Seele des Quellcodes dar.

Bau der Datenbank

Finden von Datenstrukturen

Erstellen der Objekte

Konzeption von Algorithmen

Mnemonik, Inline-Dok

Erstellen der Testumgebungen

GUI-Verfeinerung

build Feature Teams with own schedule!

Fixed Shipping Date, but target milestones!

Code Complete!

Eine Software ist wie eine Modelleisenbahn - sie wird nie fertig!

### 7.2 Codierung

**"Glühlampen brennen heller, wenn man sie vor dem Einschrauben aus der Verpackung nimmt!"**

(Unbekannt)

Vergessen Sie bei der Implementierung nicht all die unzähligen alten Tipps, die immer noch ihre Gültigkeit haben:

Die erste Regel der Optimierung heißt: "tu's nicht"

Die zweite Regel der Optimierung heißt: "wenn überhaupt, tu's erst am Schluss"

Hier ein paar Überschriften aus ["The Practice of Programming"](#) von Brian W. Kernighan & Rob Pike:

"Use descriptive names for globals, short names for locals" (dt. "Benutze beschreibende Namen für globale Variablen, kurze Namen für lokale Variablen")

"Use active names for functions" (dt. "Benutze aktive Namen für Funktionen")

"Indent to show structure" (dt. "Einrücken um die Struktur zu zeigen")

"Break up complex expressions" (dt. "Komplexe Ausdrücke aufbrechen")

"Give names to magic numbers" (dt. "Benenne Zahlen" (Konstanten, Arraygrößen, Indizes, etc.))

"Don't comment bad code, rewrite it" (dt. "Schlechten Code nicht kommentieren, sondern neu schreiben")

ANTI-Beispiele aus oben genanntem Buch zur Codierung (natürlich in C, ist schließlich von Kernighan):

```
*x+=(*xp=(2*k<(n-m)?c[k+1]:d[k--]));
```

```
child=(!LC&&!RC)?0:(!LC?RC:LC);
```

"[Code Complete](#)" von Steve C. McConnell

"[Writing Solid Code](#)" von Steve Maguire

Wir haben im Februar 2002 bei dem Review eines VisualBasic-Programms folgenden Code gefunden:

```
' Update the caption and fill in the list of options.
If CurrentUser = "xxx" Then mde_kopieren.Visible = False
If CurrentUser <> "xxx" Then mde_kopieren.Visible = False
If CurrentUser = "yyy" Then Label1.Caption = ...
```

Was sagen uns die beiden ersten Zeilen? Und die Namen der Benutzer waren auch tatsächlich hart codiert! Und Label1 ist doch auch selbst erklärend, oder?

Quellverwaltung (z.B. Microsoft Visual Source Safe) einsetzen, um saubere Teamarbeit zu ermöglichen.

Nur im Notfall während der Implementierung die Tools, die Versionen oder gar die technische Plattform wechseln. Dies bedarf schon sehr kräftiger Gründe.

### 7.3 Leitung

**"Führung ist Aktion, nicht Position."**

(Rat eines COO)

Mitarbeiter führen und beaufsichtigen

Kompetenzen delegieren

Mitarbeiter motivieren

Loben und Tadeln

Aktivitäten koordinieren

Kommunikation unterstützen

Konflikte lösen

Innovationen einführen

Projektlenkungskreise und Projektsteuerungsgremien

Pläne kontrollieren und Probleme frühzeitig signalisieren

Offenheit und Fairness

Rollen im Projektteam und Schnittstellen zwischen den Mitarbeitern definieren.

Ein Projekt wird in dieser Phase durch weitere Mitarbeiter nicht beschleunigt, sondern gebremst. Die liegt daran, dass neue Mitarbeiter sich erst in den bestehenden Code einarbeiten müssen und dabei auf die Hilfe der Veteranen des Projekts angewiesen sind. Die neuen Mitarbeiter werden zu Beginn ggf. auch die eine oder andere Änderung vornehmen, die mehr schadet als nutzt. Wenn aber der Engpass früh genug erkannt wird, lohnt sich die Einarbeitung weiterer Mitarbeiter durchaus.

"Goals Begin Behaviors - Consequences Maintain Behaviors." (dt.: "Mit Zielen beginnt Verhalten - Konsequenzen pflegen Verhalten.") ("[The One Minute Manager](#)" by Kenneth Blanchard, Ph.D. & Spencer Johnson, M.D.)

["What the Boss Should Know"](#)

### 7.4 Iterationen

**"Kein Mensch ist so beschäftigt, dass er nicht die Zeit hat, überall zu erzählen, wie beschäftigt er ist."**

(Robert Lembke)

Die Versionskennungen bei Software werden typischerweise als v.r.r dargestellt, wobei v die Nummer der Version und r die Nummer der Revision ist. Wenn also Fehler behoben wurden und vielleicht einige Funktionen aufgrund der Rückmeldungen der Kunden noch mal angepasst wurden, wird eine neue Revision des Produktes auf den Markt gebracht. Einige große Hersteller haben noch kleinere Schritte eingeführt, weil der Aufwand für die Produktion einer neuen Version/Revision bei Millionenstückzahlen in verschiedenen Sprachen für mehrere Kontinente extrem aufwendig ist. So gab es von Microsoft ein Word 6.0a, in dem nur geringe Änderungen vorgenommen wurden, die in die laufende

Produktion eingeflossen sind, aber nicht zum Ankurbeln der gesamten Marketingmaschinerie geeignet waren.

Oft wird auch die Build-Nummer (Herstellungsnummer) noch mit angegeben. Diese Nummer wird immer hochgezählt, wenn die Software wieder neu generiert wird. Wenn also Entwickler eine Änderung vorgenommen haben und alle ihre Quellen übersetzt haben, brauchen sie einen neuen fertigen Stand der Software, um ihre Änderungen erst selbst testen zu können und dann in die Qualitätskontrolle geben zu können. Zur Kommunikation ist eine eindeutige Identifizierung des Stands erforderlich, und genau hierfür wird bei jeder Erstellung eines neuen Standes die Build-Nummer hochgezählt. So wird beispielsweise bei Microsoft von einem Build-Team jede Nacht ein neuer Stand von Windows generiert, der dann von einem Burning-Team (Brenn-Team) noch in der gleichen Nacht auf CDs gezogen wird. [B21]

Lieferung auf CD-ROM  
Zwischenabnahmen?

## 7.5 Änderungsanforderungen

**"Man muss sich einfache Ziele setzen, dann kann man sich komplizierte Umwege erlauben."**  
(Charles de Gaulle)

Änderungsanforderungen (englisch *Change Requests*) sind die Tücke eines jeden Software-Projekts. Wenn Sie bisher alles richtig gemacht haben und optimal im Plan liegen und endlich die Fachlichkeit und die Technik im Griff haben, kommt der Kunde jetzt garantiert mit neuen Ideen, damit Sie als Anbieter auch garantiert ins Schleudern kommen. Hier danke ich einem Kunden für den ehrlichen Spruch: "Kunde sein verdirbt den Charakter"!

Das Problem ist ja nicht die neue Idee, der geänderte Geschäftsprozess oder der bisher so zurückhaltende Anwender, der nun anhand der Prototypen die neuen technischen Möglichkeiten entdeckt hat - das Problem ist der Umgang damit! Ist die neue Idee nun wirklich neu oder ist darüber nicht schon ganz häufig gesprochen worden und das war schon immer so gemeint? Hat sich der Geschäftsprozess wirklich geändert oder haben Sie als Anbieter den Kunden nur nicht richtig verstanden? Ist der bisher so zurückhaltende Anwender wirklich nie gefragt worden und hat jetzt erst vom neuen System erfahren?

Wenn Sie als Anbieter die Anforderungen verbindlich fixiert, umfangreich analysiert und detailliert spezifiziert haben und wenn Sie all diese Prozesse in Stufen mit Zwischenabnahmen und Protokollen begleitet haben und wenn Sie wirklich nur einen einzigen Ansprechpartner haben, der seinerseits genau einen benannten Vertreter für jeden Akteur hat und wenn alle Dokumente und Prototypen all diesen Vertretern vorgeführt wurden, dann können Sie sehr genau feststellen, welche Anforderung wirklich neu oder geändert ist.

Sie können diese Änderungsanforderung "oben" in Ihren Prozess einstreuen und beobachten, welche Auswirkungen sie worauf hat: Passt die Änderungsanforderung überhaupt ins Hauptziel? Welcher Akteur stellt diese Änderungsanforderung? Welche Anforderungen kamen noch von diesem Akteur? Auf welchen Übersichts-AF und auf welche Funktions-AF wirkt sich diese Änderungsanforderung aus? Welche Detail-AF müssen alle wie geändert werden? Welche Daten werden dabei neu entstehen oder können verworfen werden? Was bedeutet dies für das Datenmodell? Welche Geschäftslogik muss geändert oder ergänzt oder verworfen werden? Welche Teile des Datenmodells und der Geschäftslogik und welche Teile in anderen tiern sind bereits implementiert und müssen wie umcodiert werden? Welche Team-Mitglieder sind also betroffen, wofür sind diese in nächster Zeit eingeplant und welchen Stundensatz haben sie? Welche neuen Testfälle zieht die Änderung nach sich und wie aufwändig ist die Qualitätssicherung jetzt? Welche Dokumentation muss in welchem Umfang geändert werden und welche Teile des evtl. bereits begonnenen Handbuchs müssen wie geändert werden?

Wie viel Zeit wird also alles in allem für diese Änderung benötigt und was kostet diese letztendlich? Wird das Produkt vielleicht sogar billiger, weil die Änderung eine Reduzierung mit sich bringt, die eintritt, bevor mit den betroffenen Stellen überhaupt irgendjemand begonnen hat? Eine Änderungsanforderung kann durchaus auch zur Minderung statt zur Mehrung führen, auch wenn dies eher untypisch ist.

Wenn Sie als Anbieter Ihrem Kunden also nachweisen können, dass es sich um eine Änderung handelt und vorrechnen können, welchen Einfluss diese Änderung auf Zeit und Kosten hat, dann kann der Kunde für sich abwägen, ob er die Änderung dennoch möchte oder nicht. Vielleicht möchte der Kunde die Änderung dann ja auch in einer späteren Version, zusammen mit einigen anderen Änderungen. Es soll nämlich schon Projekte gegeben haben, die vor lauter Änderungsanforderungen nie fertig geworden sind! Eine Änderung in einer Folgeversion ist zwar in der Regel teurer als eine Änderung in der aktuellen Version, aber eine Version später auf den Markt zu bringen als ursprünglich geplant und versprochen kann noch viel teurer werden!

Hier sei nochmals erwähnt, dass eine Änderungsanforderung natürlich umso teurer ist, je später sie in das Projekt "einschlägt". In der Pre-Analyse ist eine Änderung gedankenschnell vollzogen, in der Analyse kostet die Änderung der bereits erstellten Dokumentteile vielleicht einen Tag, nach dem Design kann es schon eine Woche sein und wenn weite Teile bereits codiert und im Handbuch verewigt sind, wird es bestimmt schon ein Monat sein. Ist das Produkt erst im Markt und muss in der nächsten Version kompatibel geändert werden, bedeutet es vielleicht ein halbes Jahr!

## 8 Test

### 8.1 Warum testen?

**"Man muss immer das Beste hoffen, das Schlimme kommt von alleine."**  
(Deutsches Sprichwort)

"Kein Produkt menschlicher Intelligenz kommt fehlerfrei zur Welt. Wir formulieren Sätze um, trennen Nähte wieder auf, setzen Pflanzen um, planen Häuser neu und reparieren Brücken. Warum sollte es uns mit Software anders gehen?" [\[B4\]](#)

Wir sollten einfach akzeptieren, dass Software per Definition fehlerhaft ist. Ein Fehler ist die Nichterfüllung einer Anforderung und damit ein Mangel im Produkt. Daher müssen Fehler nach bestem Wissen und Gewissen gesucht, gefunden und behoben werden, bevor der Kunde das Produkt erhält. Denn wir sind ja nicht nur verantwortlich für das, was wir tun, sondern auch für das, was wir nicht tun! Nur mit dieser Einstellung kann beim Kunden ein hohes Maß an Zufriedenheit sichergestellt werden - und nur ein zufriedener Kunde ist ein guter Kunde! Also muss getestet werden!

### 8.2 Testen bis zum jüngsten Tag

**"Einen Fehler begangen haben und ihn nicht korrigieren: Erst das ist ein Fehler."**  
(Konfuzius, Lunyu 15.30)

Einsatz- oder sicherheitskritische Software muss schon bei der ersten Inbetriebnahme fehlerfrei funktionieren. Dazu muss das Softwaresystem formell spezifiziert werden und verlangt eine mathematische Kontrollmöglichkeit. Hier kann man spezielle Spezifikationssprachen verwenden, die im Gegensatz zur menschlichen Sprache streng logisch aufgebaut sind. Dadurch können die Spezifikationen dann teilweise sogar automatisch verarbeitet werden. Da der Anwender in aller Regel eine solche Sprache nicht verstehen wird, muss diese formelle Spezifikation immer zusätzlich zur normalen Spezifikation erstellt werden und dient damit im Wesentlichen zum mathematischen Testen der normalen Spezifikation und ggf. zum automatischen Testen der Software. Diese Verfahren werden beispielsweise bei der Entwicklung von Software für Herzschrittmacher eingesetzt und sind äußerst aufwendig. Außerdem erfordern logische symbolische Sprachen eine enorme Konzentration und keiner hat wirklich Lust, diesen Kram zu schreiben oder gar zu lesen. Daher sind diese Verfahren nur für kleinere Softwareprodukte interessant: wie umfangreich wäre wohl die formale Spezifikation für die Cockpitsteuerung einer Boeing, wenn doch die englische schon mehrere Ordner umfasst? Erfahrungsgemäß müssen bei formeller Verifizierung die Mitarbeiter alle sechs Wochen durch ausgeruhte Kräfte ausgetauscht werden.

Darüber hinaus reicht es oft nicht, nur das Softwaresystem zu testen. Stattdessen muss das Gesamtsystem inklusive des Softwaresystems getestet werden. Eventuell versagt die Software ja erst dann, wenn die elektrische Schnittstelle wegen Feuchtigkeit durch Spritzwasser merkwürdige nicht spezifizierte Signale sendet.

Oft wird auch der Fehler begangen, unter "Testen" nur das Testen des programmierten Codes zu verstehen. Was nutzt es aber, wenn man dabei die korrekte Implementierung einer falschen Annahme überprüft? Das Testen muss also bereits beim Erstellen der Spezifikation und sogar noch früher, nämlich beim Aufnehmen der Anforderungen, erfolgen. Wenn sich beispielsweise Anforderungen widersprechen, kann auch der sicherste Programmierer kein vernünftiges Programm erstellen.

**"What the Boss Should Know":**

I've always said that the last bug will be found when the last customer dies.

### 8.3 Weiche Tests

Black-box-Test

Smoke-Test: schauen, ob es raucht ;-)

Geschmackstest ("Pepsi-Test"): Vergleich mit dem Altprodukt

## 8.4 Systematische Tests

Bug-Reporting-System - kann eine professionelle Software sein, bei kleinen Projekten aber beispielsweise auch eine Access-Datenbank oder gar nur ein Textdokument. Fehler gewichten, Bearbeitern zuordnen und mit Behebung exakt beschreiben. Fehler zeitnah beheben und nicht aufschieben. Projektplan anpassen.

Anwendertest

Feldversuch

Testfälle

Der Projektleiter eines Projekts, in dem ich 1985 teilnahm, sagte immer: "Kaum macht man's richtig, schon geht's." Diese Einstellung hilft enorm!

Lots of Code Reviews!

Zero Defects means daily Quick Tests!

Pair-Programming (XP = Extreme Programming)

"Make it fly!"

Erfahrung machen heißt: Aus Fehlern lernen! Es müssen aber nicht immer die eigenen sein; auch aus den Fehlern anderer Menschen kann man lernen. Lernen ist ein Prozess, der das ganze Leben lang nicht abreißen sollte. Konfuzius sagt: "Lernen ist eine Tätigkeit, bei der man das Ziel nie erreicht und zugleich immer fürchten muss, das schon Erreichte wieder zu verlieren."

[Extreme Programming Roadmap](#)

Schnittstellentest

Alle Einzelfehler zu Testen ist schon schwer, aber wie sieht es mit Mehrfachfehlern aus? Hat schon mal jemand die Zeit gehabt, die extrem aufwendigen Mehrfachfehlertests bei einem Softwaresystem durchzuführen?

Prüfstand: Auch der Prüfstand wird programmiert und warum sollte diese Software eigentlich besser sein, als die, die mit dem Prüfstand getestet werden soll?

Gefahr = Bedingung, die einen Unfall verursachen kann, wenn zusätzlich weitere Ereignisse eintreten

Versagen = Unvermögen einer Systemkomponente, seine vorgesehene Funktion zu erfüllen

Risiko = Kombination der Wahrscheinlichkeit eines Versagens mit dessen Konsequenzen

Unfall = Ereignis, das Schaden anrichtet

Zuverlässigkeit = Maß für die Wahrscheinlichkeit, dass die geforderten Funktionen über eine bestimmte Zeit hinweg erfüllt werden

Sicherheit = Wahrscheinlichkeit, mit der Gefahren auftauchen

Bei kritischer Software das Ausfallverhalten testen: wenn die Software oder das von der Software gesteuerte System versagt, muss ein möglichst harmloser oder neutraler Zustand erreicht werden.

Wenn beispielsweise an einer Kreuzung die Ampeln die Verbindung zur Steuerzentrale verlieren, könnten sich alle Ampeln der Kreuzung auf rot stellen. Wenn hingegen eine Ampel an einer Kreuzung total ausfällt, könnten sich alle anderen Ampeln auf rot stellen oder ebenfalls total ausstellen und die Regelung des Verkehrs den Schildern überlassen.

Banana Products = das Produkt reift beim Kunden

## 8.5 Qualität

Reduzieren Sie das Thema zunächst einfach auf diese beiden Aussagen:

- **Qualität** ist die Erfüllung aller Anforderungen.

- Wenn eine Anforderung nicht erfüllt ist, so ist dies ein **Fehler**.

ISO 8402: "**Qualität** ist die Gesamtheit von Merkmalen einer Betrachtungseinheit bezüglich ihrer Eignung, festgelegte und vorausgesetzte Erfordernisse zu erfüllen"

EN 29004 / ISO 9004 "Qualitätsmanagement und Elemente eines Qualitätssicherungssystems-Leitfaden" Absatz 14.1 definiert **Fehler** als die Situation, "dass Materialien, Bauteile oder Endprodukte die festgelegten Forderungen nicht erfüllen können." [B5].

Qualität heißt Messbarkeit

Testen ist die Methode, mit der Qualität nachgewiesen wird

Test szenarien (programmierte Test Cases) im Simulationsmodell automatisiert (mit Protokoll)

"SPICE" steht für "Software Process Improvement and Capability Determination" (Verbesserung des Softwareprozesses und Bestimmung der Leistungsfähigkeit) und stellt eine internationale Initiative zur Entwicklung eines Standards zur Beurteilung der Softwareprozesse dar. Im Wesentlichen handelt es sich um einen Satz von Dokumenten, die als Vorlage verwendet werden können und 1995 erstmalig vorgestellt wurden. Darin wird nicht nur die Softwareentwicklung betrachtet, sondern beispielsweise auch die Evaluierung von Softwareprodukten, so dass dieser Dokumentensatz für alle interessant ist, die in irgendeiner Form mit Software konfrontiert werden.

Oder wie es in [L17] ausgedrückt wird: "This framework can be used by organizations involved in planning, managing, monitoring, controlling and improving the acquisition, supply, development, operation, evolution and support of software" (Dieser Rahmen kann von Unternehmen genutzt werden, die mit Planung, Organisation, Überwachung, Kontrolle und Verbesserung von Anschaffung, Lieferung, Entwicklung, Betrieb, Fortschritt und Unterstützung von Software beschäftigt sind).

## 9 Übergang

### 9.1 Übergang der Software an den Kunden

**"Sagt der Walfisch zum Thunfisch: 'Das kannst du doch nicht tun Fisch.'**

**Sagt der Thunfisch zum Walfisch: 'Da hab ich keine Wahl Fisch.'"**

(Unbekannt)

Von einem "Launch" ("Start") spricht man eher bei einem kommerziellen Produkt (so richtig mit Handbuch im Vierfarbkarton), das in Stückzahlen hergestellt und über Distributionskanäle verkauft wird. In dem Moment, wo die Auslieferung des Produkts beginnt, spricht man auch von "Shipping" (Lieferung). Es gab mal einen sehr interessanten Vortrag auf einem der ersten Microsoft TechTalks von dem Produktmanager von Excel 3; er nannte den Vortrag "Shipping Software", weil er dies als Hauptziel einer jeden Softwareentwicklung betrachtete und damit den Entwicklern klar machen wollte, dass eben nicht die Entwicklung selbst das Wesentliche ist, sondern das Produkt, das am Ende steht!

Abnahme nach Leistungsbeschreibung (Akzeptanzkriterien der Anforderungen)

Abnahmeprotokoll mit Restpunktliste für die Nachbesserungen  
auf vereinbarte Konventionalstrafen achten

Inbetriebnahme

- direkte Umstellung
- Parallellauf
- Versuchslauf
- Pilotinstallation

Installation nur mit Administratoren des Kunden zusammen!!!

## 10 Wartung

### 10.1 Wartung

**"Und immer wenn du denkst, es geht nicht mehr, kommt von irgendwo ein Lichtlein her."**  
(altdeutsche Weisheit)

Haftung, Garantie, Gewährleistung, Sachmängel

Software IST fehlerhaft!

- Serviceplan definieren

Auch Software altert!

- Update-Stufen entwerfen

- Versionsmanagement

Helfen Sie!

- Coaching

- Training

- Support

Nachvertrag

## 11 Verweise

### 11.1 HTTP-Links

alphabetisch sortiert

- [L1] Firma [Alistair Cockburn, Humans and Technology](#)
- [L2] [Extreme Programming Roadmap](#)
- [L3] [OMG](#) (Object Management Group)
- [L4] Firma [oose](#)
- [L5] [Patterns and Software Essential Concepts and Terminology](#)
- [L6] [Patterns Home Page](#)
- [L7] Firma [Rational Software](#)
- [L8] [The Magical Number Seven](#)
- [L9] Firma [Rösch Consulting](#)
- [L10] [Scheissprojekt](#)
- [L11] [UML](#) (Unified Modeling Language)
- [L12] [Use Cases](#)
- [L13] [What the Boss Should Know](#)
- [L14] [ANSI](#)
- [L15] Firma [Microsoft](#)
- [L16] [CORBA](#)
- [L17] [SPICE](#)

### 11.2 Bücher

alphabetisch nach Titel sortiert  
mit Links nach Amazon, wo möglich

	<b>Titel</b>	<b>Autoren</b>	<b>ISBN</b>
[B1]	<a href="#">Code Complete</a>	Steve C. McConnell	1-556-15484-4
[B2]	<a href="#">Das UML-Benutzerhandbuch</a>	Grady Booch - Jim Rumbaugh - Ivar Jacobson	3-827-31486-0
[B3]	<a href="#">Deep Time</a>	Gregory Benford	0-380-97537-8
[B4]	Digitales Verhängnis - Gefahren der Abhängigkeit von Computern und Programmen	Lauren Ruth Wiener	3-89319-672-2
[B5]	Qualitätssicherung und angewandte Statistik, Verfahren 3: Qualitätssicherungssysteme	DIN-Taschenbuch 226, Beuth 1992	3-410-12761-5
[B6]	<a href="#">Iacocca</a> . Eine amerikanische Karriere	Lee Iacocca & William Novak	3-430-14937-1
[B7]	<a href="#">Lehrbuch der Objektmodellierung, Analyse und Entwurf</a>	Heide Balzert	3-827-40285-9
[B8]	<a href="#">Per Anhalter durch die Galaxis</a>	Douglas Adams	3-453-14697-2
[B9]	<a href="#">The Hitchhiker's Guide to the Galaxy</a>	Douglas Adams	0-517-14925-7
[B10]	<a href="#">The Mythical Man Month</a>	Frederick P. Brooks, Jr.	0-201-835959

[B11]	<a href="#">The One Minute Manager</a>	Kenneth Banchard, Ph.D. & Spencer Johnson, M.D.	0-688-01429-1
[B12]	<a href="#">The Practice of Programming</a>	Brian W. Kernighan & Rob Pike	0-201-61586-X
[B13]	<a href="#">UML konzentriert</a>	Martin Fowler und Kendall Scott	3-8273-1617-0
[B14]	<a href="#">Writing Effective Use Cases</a>	Alistair Cockburn	0-201-70225-8
[B15]	<a href="#">Writing Solid Code</a>	Steve Maguire	1-556-15551-4
[B16]	<a href="#">Software Engineering Economics</a>	Barry W. Boehm	0-138-22122-7
[B17]	Analysemuster - Wiederverwendbare Objektmodelle	Martin Fowler	3-8273-1434-8
[B18]	Best of Booch	Grady Booch	0-13-739616-3
[B19]	<a href="#">Cyber Rules</a>	Thomas M. Siebel, Pat House	0-385-49412-2 3478245206
[B20]	<a href="#">Entwurfsmuster</a>	Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides	3-89319-950-0 3827318629
[B21]	<a href="#">Der Krieg des Codes. Wie Microsoft ein neues Betriebssystem entwickelt. (Englischer Titel: Showstopper)</a>	G. Pascal Zachary	345511038X