# 4. Teams, Tools und deren Bewertung

Ohne kompetente und gut ausgebildete Software- Ingenieure kann kein Software Projekt erfolgreich durchgeführt werden. Aber auch die richtigen Leute alleine sind kein Garant für ein erfolgreiches Projekt. Das Teamumfeld muss so sein, dass die Teammitglieder produktiv und kooperativ arbeiten können. Wir lernen in der ersten Hälfte dieses Kapitels verschiedene Merkmale erfolgreicher Teams kennen.
Software Ingenieure benötigen zwei Arten von Werkzeugen:

1. analytische Werkzeuge, wie die stufenweise Verfeinerung und die Kosten- Nutzen Analyse
2. Software Werkzeuge, also Produkte und Werkzeuge, mit deren Hilfe der Entwickler die Software entwickelt und wartet. Software Werkzeuge, welche
   bei der Entwicklung behilflich sind
   die Dokumentation unterstützen
   das Vorgehen leiten
   das systematische Testen ermöglichen oder erleichtern
   ...
   Diese Werkzeuge fasst man in der Regel zusammen unter dem Begriff CASE = Computer Aided Software Engineering

## *4.1. Team Organisation*

Die meisten Software Produkte sind zu komplex, um von einer Person alleine erstellt zu werden. Das hat zur Folge, dass mehrere Entwickler sich für die Entwicklung und Wartung des Produktes, zusammen finden müssen, also ein Team bilden müssen.

### 4.1.1. Beispiel

Sie arbeiten in einer Informatikfirma, die Spezialsoftware für Treuhandfirmen erstellt und vertreibt. Die Firma plant die Software von Grund auf neu zu entwickeln, unter Einsatz moderner Werkzeuge, Datenbanken und neuer Methoden.

#### 4.1.1.1. Anforderungsphase

Da die Firma sich zwar im Informatik Bereich bestens auskennt aber wenig Fachwissen über Treuhandbetriebe hat, setzt sie ein Team zusammen, zur Definition der Anforderungen. Dieses Team besteht aus einem Treuhänder, der die Anforderungen bestens kennt, sowie zwei internen Mitarbeitern und einem externen Moderator.
Die Informatikfirma masst sich also nicht an, die Anforderungen der Branche gut genug zu kennen.

#### 4.1.1.2. Implementationsphase

Da die Firma erst kürzlich die OO Methodik "Rational Unified Process" eingeführt hat, engagiert sie einen externen Berater, der die Methode kennt und das Entwicklerteam beraten kann.

Es stellt sich heraus, dass das Projekt vermutlich 12 Monate dauern wird. Da die Firma drei Programmierer hat, behauptet das Management, die Software wäre in vier Monaten marktreif.

Die Rechnung des Managements ist natürlich falsch! Falls eine Frau neun Monate benötigt, um ein Kind zu produzieren, schaffen es neun Frauen nicht in einem Monat.

Der Schlüssel liegt in der Arbeitsteilung: es gibt aber auch Arbeiten, die kann man nicht aufteilen!

### 4.1.2. Arbeitsteilung in der Software Entwicklung

In der Software Entwicklung ist die Arbeitsteilung auch nur bis zu einem bestimmten Grad möglich. Dabei muss man aber ein Grundproblem der Teambildung beachten: die Teamkommunikation!

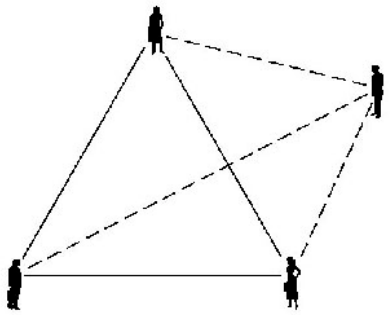
#### 4.1.2.1. Beispiel

Team A erstellt die Module M1 und M2.
Team B erstellt die Module M3 und M4.
Damit das Gesamtsystem funktioniert, benötigt das Integrationsteam C die Module gemäss der ursprünglichen Spezifikation.
Im Laufe der Entwicklung von Modul M1 stellt sich heraus, dass M2 aus M1 einen zusätzlichen Parameter benötigt. Das Team A muss die Schnittstelle zwischen Modul M1 und Modul M2 ändern. Team A geht davon aus, dass diese Änderung keinen Einfluss hat auf die anderen Module. Es zeigt sich, nach einigem Programmieraufwand, dass es sinnvoll ist, die Parameterreihenfolge von Modul M2 umzudrehen. Auch hier geht Team A davon aus, die Änderungen voll im Griff zu haben.
Was passiert?
Team C, das Integrationsteam bekommt ein grösseres Problem:
die Schnittstelle zwischen Modul M1 und M2 wird auch von andern Modulen genutzt; die Änderung der Parameterreihenfolge von Modul M2 ist nicht erkennbar, weil die vertauschten Parameter vom gleichen Typ sind. Team C weiss nicht warum das System falsche Ergebnisse produziert.

Das eben beschriebene Problem hat nichts mit technischer Inkompetenz zu tun! Es handelt sich schlicht um ein Managementproblem! Es ist die Aufgabe des (Projekt-) Managements ein kreatives Umfeld zu schaffen, in dem sich die Teams voll entfalten können und produktiv sein können.

Ein weiteres Problem ist die Team- Kommunikation, die wir oben schon angesprochen haben. Betrachten wir folgendes Kommunikationsdiagramm:

am Anfang besteht das Team aus drei Teammitgliedern. Daraus resultieren drei mögliche Kommunikationswege (A mit B, A mit C, B mit C).



Sobald ein weiteres Teammitglied hinzu kommt, resultieren sechs Kommunikationswege, also doppelt so viele wie vorher!

Was heisst das?
Wenn wir ein Projektteam verstärken wollen, zum Beispiel wegen Verzögerungen, erhöhen wir den Kommunikationsbedarf gewaltig. Das Team wird ineffizienter!

Diese Tatsache, dieses Prinzip wird auch als "Brook's Gesetzt" bezeichnet, der dieses Phänomen beim Entwickeln des IBM Betriebssystems OS/360 entdeckte.
Teams werden vor allem während der Implementationsphase benötigt, weil dort der Grossteil der Arbeit anfällt. Wir werden uns daher in diesem Kapitel vor allem auf diese Phase und deren Probleme konzentrieren.

Teambildung geschieht in Entwicklungsprojekten in unterschiedlichen Formen. Die zwei Grenzfälle sind:

1. Democratic / Demokratisches Team
2. Chief Programmer / Leit- Programmierer Team

Wir beschreiben im Folgenden beide Organisationsformen und bewerten diese anschliessend.

### *4.2. Democratic Team Approach*

Die demokratische Teamform wurde zum ersten Mal von Weinberg 1971 in einem Buch *"The Psychology of Computer Programming"* beschrieben. Das grundlegende Konzept dieser Organisationsform wird als "*egoless programming"* bezeichnet. Weinberg hat beobachtet, dass sich viele Programmierer voll mit ihrem Werk, dem Programmcode identifizieren. Einige nennen die Module, die sie erschaffen, nach sich oder einer nahe stehenden Person. Dieser Programmcode ist quasi eine Erweiterung des Egos des Programmierers angesehen werden. Die Konsequenz ist, dass der Programmierer in seinem Programm sicher möglichst wenig Fehler findet, da er ja sein Ego nicht verlieren möchte. Er hütet also seinen Programmcode vor unerwünschtem Zugriff.

Weinberg sieht den Ausweg aus diesem Problem in einem *Ego- losen* Programmieren. Das gesamte soziale Umfeld muss neu strukturiert werden. Jeder Programmierer muss seine Kollegen auffordern , in seinem Programm Fehler zu finden. Der Fehler darf nicht als etwas negatives dargestellt werden, sondern als etwas normales! Der Hinweis auf einen Fehler muss als Ratschlag, nicht als Kritik aufgefasst werden. Das Team als Ganzes muss ein "Gruppenego" entwickeln. Das Team als Ganzes ist "Besitzer" der Programme, nicht der individuelle Programmierer.

Ein Team aus 10 Ego- losen Programmierern bildet zusammen ein *Demokratisches Team.*

Weinberg weist darauf hin, dass das Management eines solchen Teams sehr schwierig sein kann, speziell wenn einer der Ego- losen Programmierer befördert wird! Wesentlich ist der gegenseitige Respekt und die Teamverantwortung, das Teambewusstsein als der "Leim", der das Team zusammen hält.

Weinberg beschreibt ein demokratisches Team, dessen Team Manager das Firmen Management einen hohen Bonus bezahlte, weil das Team ein sehr umfangreiches und komplexes Projekt zum Erfolg führte. Der Projektleiter verteilte den Bonus gleichmässig unter allen Teammitgliedern. Das Management konnte dieses Verhalten nicht verstehen. Bald darauf verliess das gesamte Team die Firma.

Untersuchen wir jetzt die Vorteile und Nachteile eines solchen Teams.

### 4.2.1. Analyse der demokratischen Team Organisation

Hauptvorteil eines demokratischen Teams ist die positive Einstellung, die Fehler in den Programmen zu finden. Je mehr Fehler gefunden werden, desto glücklicher ist das Team. Diese positive Einstellung führt auch zu einem schnelleren Aufdecken von Fehlern. Grund dieses Verhaltens ist der Stolz des Teams auf korrekte Programme.

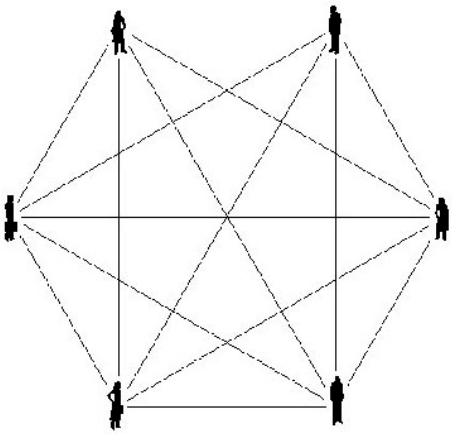
Für das Management ist ein solches Team eher etwas Exotisches!

Es ist auch nicht sehr einfach für einen erfahrenen Programmierer sich Ratschläge von einem Anfänger geben zu lassen.

Gemäss Weinberg können solche Teams nicht auf Druck von aussen entstehen. Falls sie entstehen, sind sie in der Regel extrem effizient. Es wird vermutet, dass solche Teams vor allem im Forschungsumfeld entstehen können. Oft wandelt sich ein demokratisches Team nachdem es die schwierigen Probleme gelöst hat und nun die alltäglichen Probleme lösen muss, in ein normales Team, mit hierarchischen Strukturen.

### *4.3. Die klassische Chief Programmer Organisation*

Betrachten wir ein weiteres Kommunikationsdiagramm, diesmal ein Team bestehend aus sechs Mitgliedern.

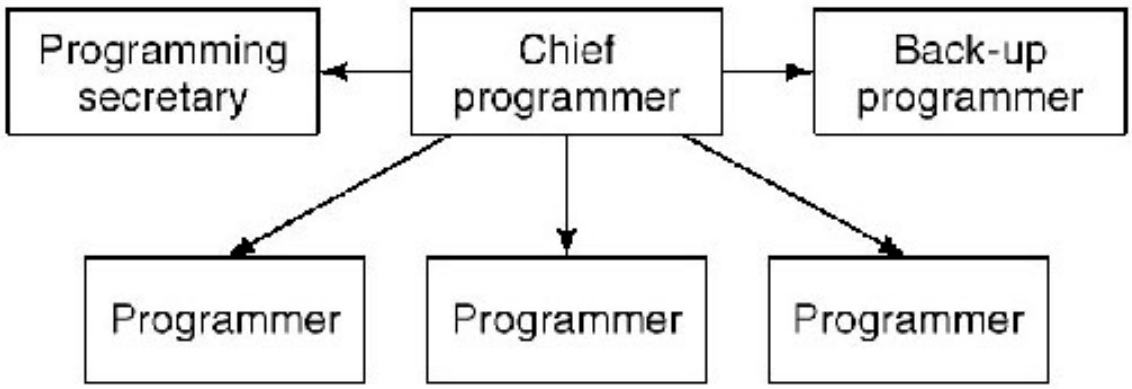


Dieses Team kennt somit insgesamt 15 Zwei-Personen-Kommunikations Beziehungen.
Aber daneben existieren auch Kommunikations-Subsysteme mit drei Kommunikations- Partnern.
Insgesamt kann man 57 mögliche Kommunikations- Muster definieren!

Diese Komplexität ist der Hauptgrund warum 6 Programmierer nicht 6 mal weniger für die Fertigstellung benötigen wie ein einzelner Programmierer. Der Kommunikations-/ Koordinations- Aufwand ist schlicht zu gross.

Betrachten wir nun ein anderes Kommunikationsdiagramm, ebenfalls mit sechs Kommunikations- Partnern, nur eben anders angeordnet.

Diese Struktur besitzt lediglich fünf Kommunikationsverbindungen! Dieses Schema bildet die




Basis des "*Chief Programmer Team*".

Zwei evident anders als beim Ego- losen Team behandhabten Aspekte dieser Organisationsform sind:

1. *die Spezialisierung*
2. *die Hierarchie*

Die Organisationsform wurde 1971 vorgeschlagen, von Mills, der damals ein komplexes Projekt für eine New Yorker Zeitung zu leiten hatte.

Sie besteht aus einem Chief Programmer, dessen Assistenten (Backup Programmer), einer Sekretärin und einem bis drei Programmierern.

Der *Chief Programmer* ist ein erfolgreicher Manager und ein hervorragender Techniker. Er legt die Architektur des Informationssystems fest. Er beschreibt, wie kritische Programmteile aus zu sehen haben.

Die andern Programmierer sind für den Design und die Implementierung von bestimmten Modulen zuständig. Die Kommunikation zwischen den Programmieren geschieht über den Chief, da er die höchste technische Instanz ist. Der Chief Programmer führt auch alle Reviews durch. Der Chief Programmer ist für jede Zeile Programmcode verantwortlich.

Der Chief Programmer hat noch einen Assistenten, den *Backup Programmer* , der immer dann einspringt, wenn der Chief ausgelastet oder in den Ferien oder krank oder ... also nicht verfügbar ist. Er kennt das Projekt genau so gut wie der Chief. Die Aufgabenteilung zwischen dem Chief und dem Backup Programmer sieht so aus, dass der Chief sich eher um Architekturfragen, der Backup sich auf das Testen konzentriert.

Die Hauptaufgaben der *Sekretärin* besteht nicht in einem Sekretärinnen- Job, sondern sie trägt die Verantwortung für die Projektbibliotheken, die Dokumentation des Projektes und der Programme, inklusive den Testdaten.

Die *Programmierer* sind hoch fokusierte Spezialisten, die sich auf die Programmierung konzentrieren. Die Sekretärin nimmt ihnen alle unangenehmen Aufgaben ab. Zu jener Zeit bestand das Programmieren auch noch vor allem aus Lochkarten stanzen.

### 4.3.1. Das *New York Times* Projekt

Das Chief Programmer Konzept wurde zum ersten Mal in einem Projekt der IBM für die New York Times eingesetzt. Jeder der diese Zeitung und das Umfeld in New York kennt, weiss was ein Misserfolg zur Folge hätte.

Bei dem Informationssystem ging es um die Verwaltung aller Zeitungstexte, also sicher eines strategischen Informationssystems.

Die Fakten des Projektes sind imposant:
83'000 Zeilen Programmcode (LOC : Lines of Code) wurden in 22 Monaten geschrieben, total 11 Personenjahren.
Nach dem ersten Projektjahr waren lediglich die Datei- Verwaltungsprogramme erstellt, mit insgesamt 12'000 Programmzeilen. Die meisten Programmzeilen wurden in den letzten 6 Monaten geschrieben.
In den ersten 5 Wochen nach der Einführung wurden lediglich 21 Fehler gefunden. Weitere 25 Fehler wurden im Verlaufe des ersten Jahres gefunden. Pro Programmierer wurde ein Fehler auf 10'000 Zeilen Programmcode gefunden pro Programmierjahr.
Das Dateiverwaltungssystem war 20 Monate in Betrieb bevor ein einziger Fehler gefunden wurde. Die meisten der 200 bis 400 zeiligen Programme funktionierten auf Anhieb.

Für IBM handelte es sich um ein Prestigeprojekt. Aber warum war das Projekt so erfolgreich?

Einer der Gründe für den Erfolg war, dass am Projekt fast nur extrem hoch qualifizierte Mitarbeiter der IBM beteiligt waren:
1. die Entwickler des Betriebssystems, des Compilers, der Job Control Sprache
2. IBM hatte als Backup Programmierer die Spitzenentwickler abgestellt

3. der Chief Programmer war ein sogenannter Superprogrammierer, ein Programmierer, dessen Produktivität etwa fünf mal höher ist, als die eines normalen Programmierers. Er war zudem ein exzellenter Manager und eine echte Führernatur, der die Mitarbeiter begeistern konnte.

Solche Teams hängen kritisch vom Chief Programmer ab. Falls dieser nicht wirklich weit überdurchschnittlich ist, misslingt das Projekt.

### 4.3.2. Unmöglichkeit des klassischen Chief Programmer Teams

Die Qualifikation des Chief Programmers muss so hoch sein, dass es fast unmöglich ist, solche "Supermenschen" zu finden.

Die Backup Programmierer, die nicht wesentlich tiefer qualifiziert sein dürfen, findet man ebenfalls kaum.

Sekretärinnen für Chief Programmer Teams sind ebenfalls sehr schwierig zu finden, da sie sich mit den notorischen Gegnern jeder administrativen Arbeit, den Programmieren beschäftigen muss.

Zusammenfassend muss man beide Extremvarianten "Demokratisches Team" und "Chief Programmer Team" als zu extrem und kaum realisierbar ansehen.

Diese Organisationsform ist kaum geeignet, Teams mit mehreren 100 Programmieren sinnvoll zu managen.

### *4.4. Es gibt auch noch anderes als Chief Programmer und Democratic Teams*

Demokratische Teams haben den Vorteil, eine positive Einstellung zu haben und die Fehlersuche als etwas Positives anzusehen.

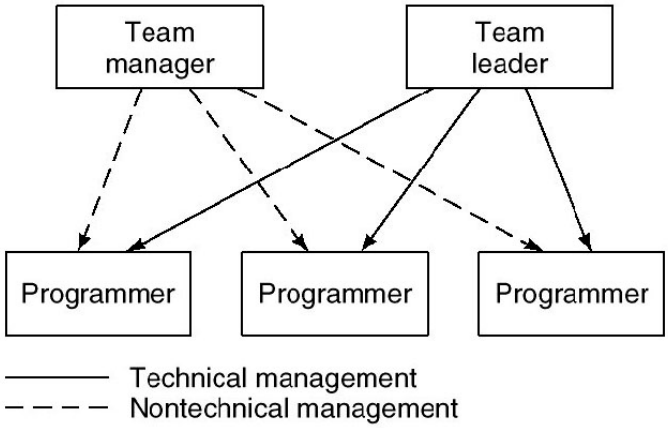
Viele Firmen kombinieren ein Chief Programmer Team ähnliches Organigramm mit einer Review Technik analog zum democratic programming team.

Die Doppelbelastung des Chief Programmers (Technik & Management) muss in der Regel entschärft werden, indem die eine oder die andere Aufgabe im Vordergrund steht.

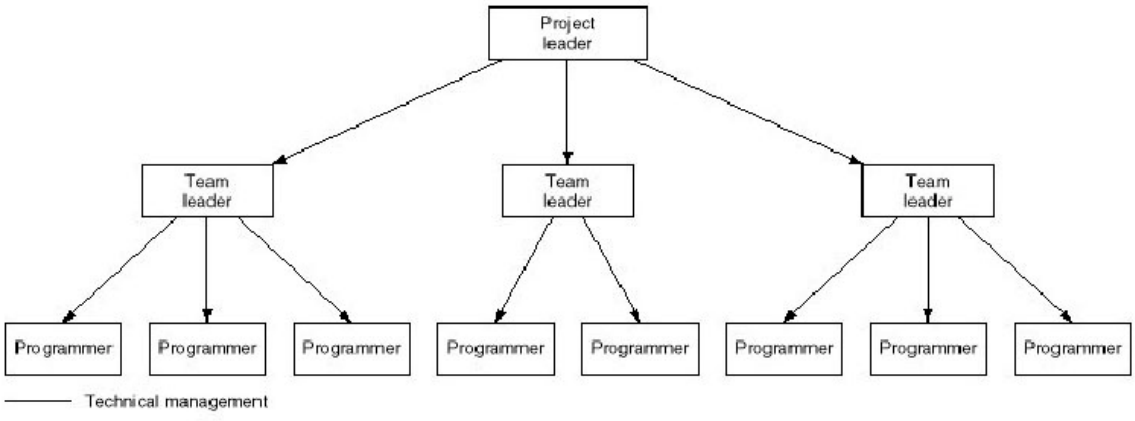
Die Aufgabe wird also aufgeteilt: es gibt einen Projektleiter und einen Projektmanager.

Der Manager trägt die Budgetverantwortung, er kümmert sich auch um legale Fragen, Anstellungsfragen usw.

Vor Projektbeginn müssen die Verantwortungsbereiche des Projektleiters und des Projektmanagers klar festgelegt werden.

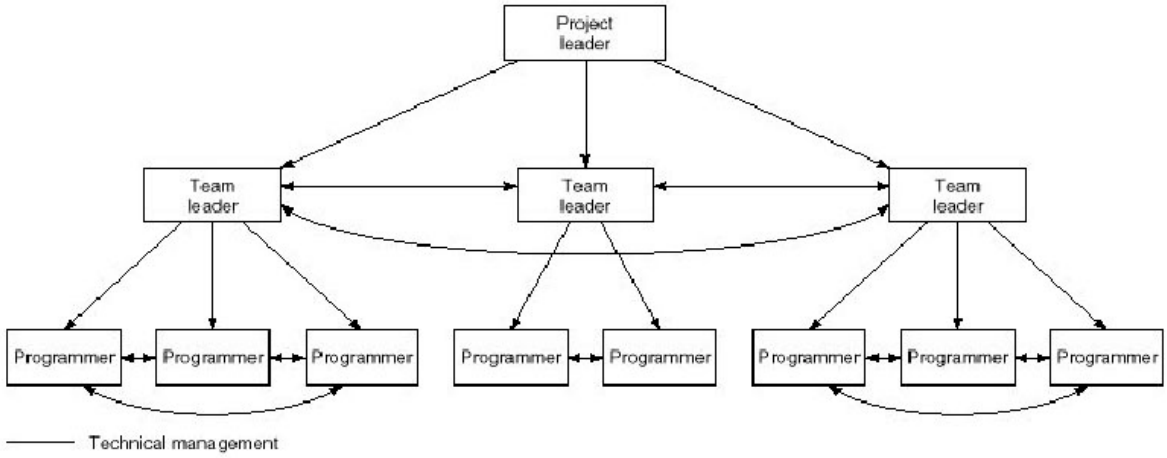



Falls das Projekt komplexer ist, funktioniert auch diese Aufteilung nicht mehr. Man muss dann eine weitere Mischform anwenden:




Hier wird das Team in mehrere kleinere Teams zerlegt. Die kleineren Teams sind analog zu einem Chief Programmer Team aufgebaut

Auch für das democratic programmer Team gibt es "Grossvarianten":




In dieser Form werden Entscheide, die technische Fragen betreffen, soweit wie möglich an die Programmierteams delegiert.

Warum ist diese Form eher selten?
Hauptgrund ist das fehlende Verständnis des Managements! Es kann sich schlicht nicht vorstellen, dass eine solche Organisationsform überhaupt funktioniert.

### *4.5. Synchronize-and-Stabilize Teams*

Microsoft hat etwas mehr als 200 Programmierer. Daher muss sich Microsoft Gedanken darüber machen, wie diese hoch qualifizierten Mitarbeiter motiviert werden können und zu einer möglichst hohen Produktivität angeleitet werden können.

Der Lebenszyklus von Microsoft wurde bereits besprochen und Sie hatten die Aufgabe einen vertiefenden Artikel darüber zu lesen.

Die Grundidee ist aber auch hier : kleine Teams, die sich möglichst selber organisieren und für sich eine grosse Autonomie aber auch Verantwortung geniessen.

Es gibt, wie Sie gelesen haben, einige Grundregeln:
1. Fehler, die bei der Synchronisation auftreten, müssen sofort beseitigt werden!
2. der Programmcode muss zur vereinbarten Zeit abgelegt werden

Microsoft, als Software Gigant, wendet sehr unterschiedliche Organisationsformen an, nicht zuletzt weil die Kleinteams sehr unabhängig sind.

### *4.6.*

### *Zusammenfassung : Teamformen, Projektorganisation*

Fassen wir die verschiedenen Ausprägungen der Teamorganisation zusammen:

| **Team Organisation** | **Stärken** | **Schwächen** |
|---|---|---|
| Democratic Teams | qualitativ hochstehender Code; Positive Einstellung zum Finden von Fehlern; gut geeignet zum Lösen komplexer Probleme. | kann nicht von aussen verlangt werden (ich kann niemandem befehlen positiv zu sein) |
| Chief Programmer Teams | grosser Erfolg im New York Times Projekt | unpraktisch (Qualifikationsprobleme) |
| modifiziertes Chief Pgm T | viele Erfolge | kein Vergleich mit dem Grosserfolg der NYTimes |
| moderne Prog. Teams | Team Manager + Team Leader Erfolgreich auch bei 100+Pgm. | Verantwortungsbereiche müssen klar definiert werden |

## *4.7. Stufenweise Verfeinerung*

Stufenweise Verfeinerung ist ein Problemlösungsverfahren, eine Vorgehensweise. Im Ansatz besteht die Methode der stufenweise Verfeinerung aus einem Verlagern von Entscheiden auf einen späteren Zeitpunkt. Wir zerlegen das Problem, lösen aber das Problem noch nicht. Zu einem späteren Zeitpunkt müssen wir dann die Teilprobleme lösen und zur Gesamtlösung zusammen setzen.

Stufenweise Verfeinerung ist auch aus einem andern Grund wesentlich: gemäss einem Psychologen kann sich der Mensch mit 5 +/- 2 *chunks* (Informations-Brocken) gleichzeitig befassen. Software Systeme haben natürlich weit mehr als 7 Problemkreise. Wir müssen also komplexe Probleme in kleinere zerlegen.

Wir wollen im Folgende ein Mini- Problem durch stufenweise Verfeinerung in Teilprobleme zerlegen und so das Problem lösen.

### 4.7.1. Problemstellung

Wir möchten eine sequentielle Datei verwalten, also neue Datensätze einfügen, bestehende Datensätze verändern und Datensätze löschen.

Das Problem ist einfach, auch die Lösung. Wir brauchen uns also nicht lange mit der Problembeschreibung aufzuhalten, sondern können uns auf die stufenweise Verfeinerung konzentrieren.

Formalisierung der Problemstellung:

Wir haben drei Transaktionstypen:

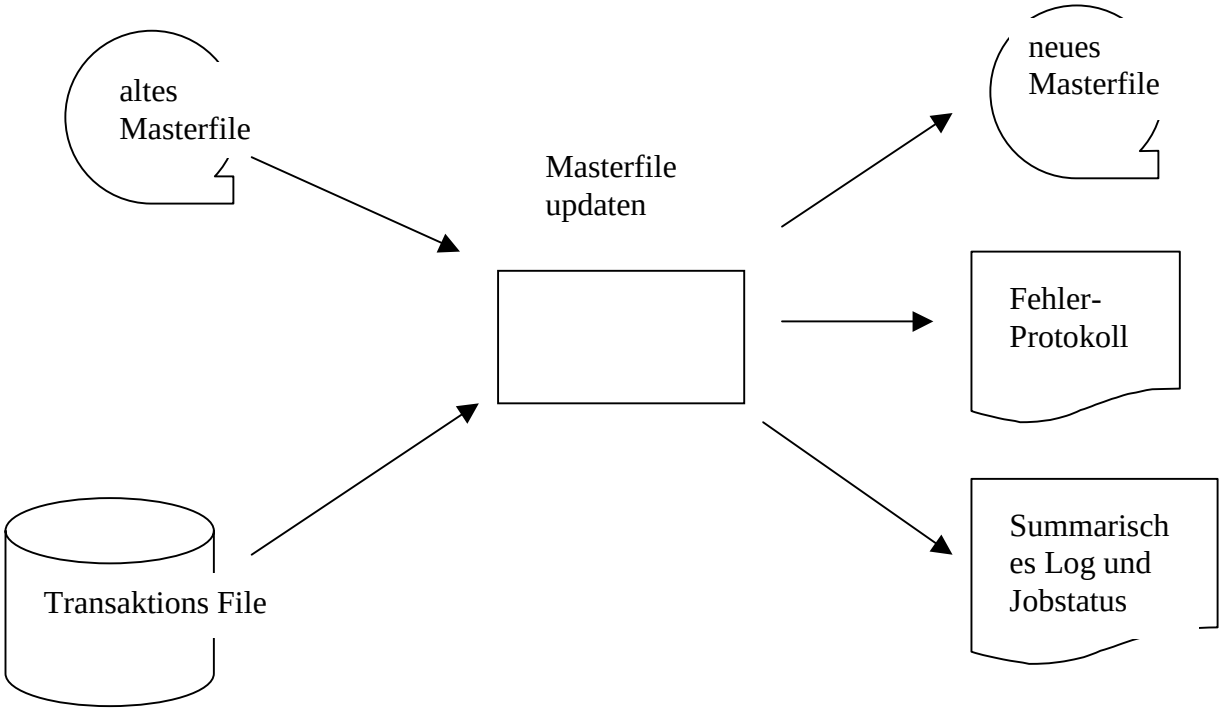
- Typ 1 : INSERT (einen neuen Datensatz einfügen)
- Typ 2 : MODIFY (einen existierenden Datensatz verändern)
- Typ 3 : DELETE (einen bestehenden Datensatz löschen)

Und hier ein typisches Logfile beziehungsweise Transaktionsfile:

| **Transaktions Typ** | **Name** | **Adresse** |
|---|---|---|
| 3 | Heinrich | Baumstrasse 12 |
| 1 | Blum | Aareweg 98 |
| 3 | König | Löwenstrasse 12 |
| 2 | Hunziker | Chaletweg 34 |
| **2** | Hermann | Schlossweg 2 |

Die Transaktionen sollen sortiert werden und zwar nach Name, dann nach Transaktionstyp. Damit ist gewährleistet, dass INSERT vor MODIFY vor DELETE ausgeführt wird, falls alle drei Transaktionstypen auf eine und die selbe Adresse angewendet werden.

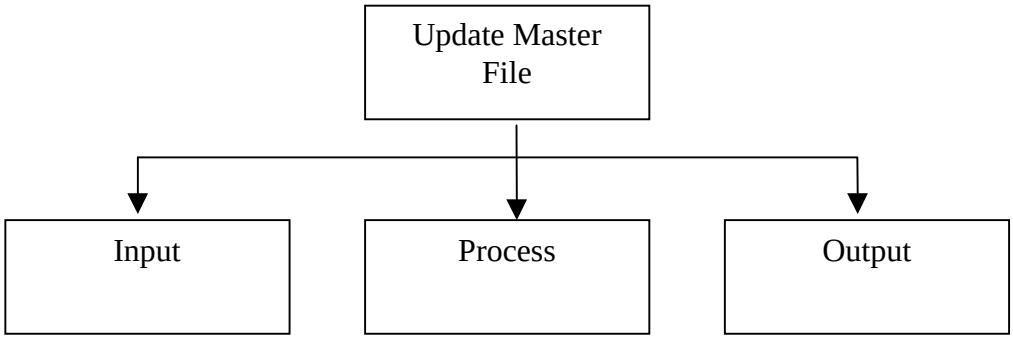
Wir haben unsere Lösung in ein bestehendes System hinein zu entwickeln:
Das System hat folgende Struktur:




Unser System hat also zwei Eingabefiles:
1. Das alte Masterfile
2. Das Transaktionsfile

Und drei Ausgabefiles / Ausgaben:
1. Das neue Masterfile
2. Der Ausnahme- Bericht / das Fehler- Protokoll
3. Einen summarischen Job- Bericht / ein Job Log, mit kurzen Angaben zum Job und Programmcodes

Wie soll unser System aussehen?
Nun, auf dieser Betrachtungsebene nennen wir unsere Blackbox einfach UpdateMasterfile.
Als erstes zerlegen wir diese Box in drei Teile:
1. Eingabe
2. Verarbeitung
3. Ausgabe



Was macht PROCESS?
Spielen wir den Ablauf einfach einmal durch.

Betrachten wir die Beispiel- Dateien:

Transaktions-File

| | |
|---|---|
| 3 | Braun |
| 1 | Herrman |
| 2 | Herren |
| 3 | Herren |
| 1 | Schmid |

Altes Master-File

| |
|---|
| Abel |
| Braun |
| Jakob |
| Herren |
| Schmid |
| Türler |

Neues Master-File

| |
|---|
| Abel<br>Herrman |
| Jakob |
| Schmid |
| Türler |

Ausnahme-Report

| |
|---|
| Schmid |

Was geschieht im Detail?
Braun wird von Transaktionsfile gelesen; Transaktion : löschen
Abel wird vom alten Master-File gelesen
Die Schlüssel von Abel und Braun sind unterschiedlich, also wird Abel ins neue Masterfile geschrieben.

Braun wird vom alten Master-File gelesen und mit dem Transaktions- Record (Braun) verglichen.
Da beide gleich sind, wird Braun gelöscht, dh nicht ins neue Master- File geschrieben.

Das Transaktions-File wird erneut gelesen (1 Herrman ; einfügen).
Das alte Master- File wird gelesen (Jakob).
Herrman kommt vor Jakob, wird also ins neue Master- File geschrieben. Herrman steht jetzt dort, wo früher Braun stand.

Herren ist ein spezieller Fall: Herren wird vom Transaktions- File gelesen (Update) und vom alten Master- File.
Herren wird also verändert. Aber nachher wird vom Transaktions- File nochmals Herren gelesen (Delete) und somit gelöscht. Herren fehlt also im neuen Master- File.

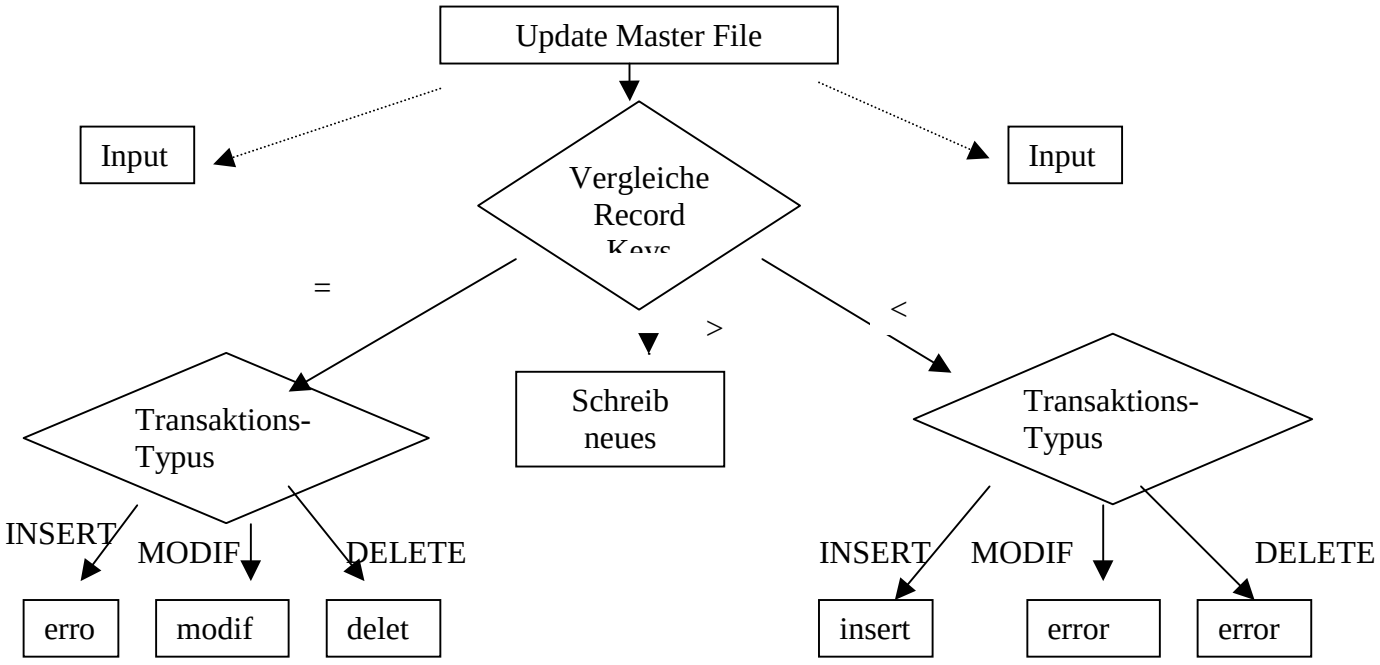
Damit sind wir in der Lage in Pseudocode unser UpdateMasterFile zu verfeinern:

| Key-Vergleich | Aktivität | | |
|---|---|---|---|
| Transaktions-Record-Key | INSERT | : | nicht möglich ; FEHLERMELDUNG |
| = | MODIFY | : | Masterfile Record updaten |
| Masterfile-Record-Key | DELETE | : | Masterfile Record löschen |

Transaktions-Record-Key

| | | | |
|---|---|---|---|
| > | kopiere alten Masterfile Record ins neue Masterfile | | Masterfile-Record-Key |
| Transaktions-Record-Key | INSERT | : | Transaktions-Record ins neue Masterfile schreiben |
| < | MODIFY | : | nicht möglich; FEHLERMELDUNG |
| Masterfile-Record-Key | DELETE | : | nicht möglich; FEHLERMELDUNG |

**Oder als "Flow Chart":**




Als nächstes müssen wir uns mit den Eingabe und Ausgabe Routinen befassen.

Wir lassen dies hier weg, da das Vorgehen nach der ersten Zerlegung eigentlich klar sein dürfte.

Was sind die Stärken und Schwächen der stufenweise Verfeinerung?

Der Software Engineer kann sich auf ein klar definiertes Problem konzentrieren, braucht sich nicht um alle Details zu kümmern und kommt dadurch zu einer modularen Lösung seines Problems.

## *4.8. Kosten- Nutzen Analyse*

In der Regel muss ein Projektantrag eine Kosten- Nutzen Abschätzung beinhalten.
Wie führt man eine solche Abschätzung durch.

Zuerst muss man unterscheiden zwischen Einmalkosten (Investitionskosten) und laufenden Kosten (Betriebskosten), internen und externen Kosten.

### 4.8.1. Investitionskosten und Betriebskosten

| Kosten Träger | Investitionskosten externes Entgelt | Interne Kosten | Betriebskosten Externes Entgelt | Interne Kosten |
|---|---|---|---|---|
| Applikations-Software | Lizenzen<br>Anpassungen<br>Installation | Anpassungen | Wartung<br>Erweiterungen | Betreuung<br>Erweiterungen |
| Hardware | Komponenten<br>Installation | | Wartung<br>Ausbauten | |
| Systemsoftware | Lizenzen<br>Installation | | Wartung<br>Ausbauten | |
| Kommunikation | Datennetz<br>Netz-Zugangsgeräte | | Benutzungs-Gebühren | |
| Infrastruktur | Bauten<br>Mobiliar | Umbau<br>Installation | | Raumkosten |
| Organisation | Beratung<br>Unterstützung<br>Projekt-Management | Ausführung<br>Projekt-Management | | |
| Einführung | Schulung<br>Unterstützung | Schulung<br>Datenaufbau<br>Abnahme | | |
| Nebenkosten | Datenträger<br>Diverses | | Verbrauchsmaterial<br>Diverses | |
| Nutzung | | | Betreuung<br>Schulung | Operating<br>Betreuung |

### 4.8.2. Wirtschaftlichkeit

Es gibt sehr unterschiedliche Investitionsrechnungsmethoden. Wir wollen aber nicht im Detail darauf eingehen.

Für eine Abschätzung reicht oft folgendes Verfahren:

#### 4.8.2.1. Abschätzung des Break Even Punktes

Investitionskosten + BreakEvenPeriode * BetriebskostenNeu = BreakEvenPeriode * BetriebskostenAlt

Wobei BetriebskostenAlt und BetriebskostenNeu jeweils pro Zeiteinheit berechnet sind (zum Beispiel pro Monat).

Oder:

BreakEventPeriode = Investitionskosten / (BetriebskostenAlt - BetriebskostenNeu)

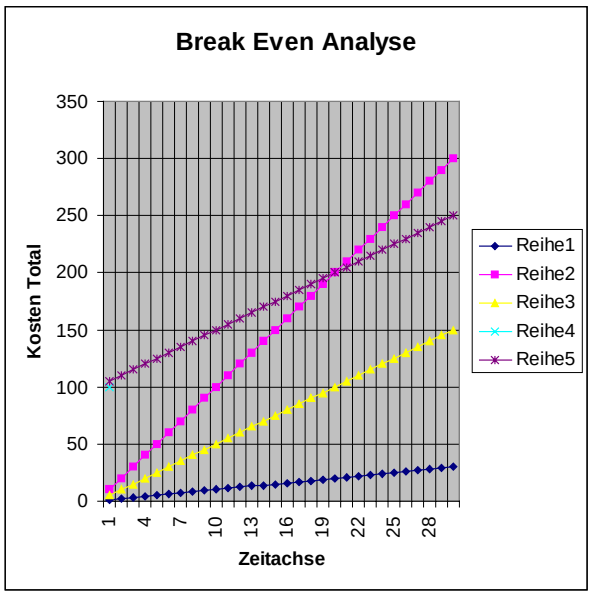
#### 4.8.2.2. Beispiel

Betriebskosten alt : 10'000.-- / Monat; Betriebskosten neu : 5'000.-- / Monat;
Investitionskosten : 100'000.--

BreakEven dh Zeitpunkt, ab dem die neue Lösung insgesamt weniger kostet als der Weiterbetrieb der alten Lösung:

BEP = 100'000 / (10'000-5'000) [Monate] = 100'000/5'000 Mt = 100/5 Mt = 20 Monate

Dh die Kosten für die Ablösung der alten Lösung (100'000.--) machen sich nach 20 Monaten Betrieb bezahlt, weil die laufenden Kosten der neuen Lösung wesentlich tiefer sind als die laufenden Kosten (Betriebskosten) der alten Lösung.




Graphisch :

Wir sehen auf dem Chart, dass die Kosten Total für die neue Lösung (Reihe 5) und die Kosten Total für das bestehende System (Reihe 2) gleich sind (Kreuzungspunkt der Geraden).

Ab diesem Zeitpunkt würde die Firma für die neue Lösung (kumuliert) weniger ausgeben als für die alte Lösung.

***4.9.***

***CASE : Computer Aided Software Engineering***

Viele Software Entwickler verwenden CASE Tools lediglich für die Dokumentation der Software.

CASE Tools sind jedoch viel mächtiger und können unterschiedliche Konsistenz-Prüfungen durchführen. Design der Software mit Hilfe von CASE Tools hat sich speziell im Datenbank Bereich als sehr effizient erwiesen. Die Datenmodelle werden mit Hilfe eines CASE Tools erfasst bezw. erstellt ; das Tool generiert dann ein Skript, mit dem eine Datenbank angelegt oder modifiziert werden kann. Besteht die Datenbank bereits, dann kann mit Hilfe des Tools ein Datenmodell erstellt werden, Modifikationen durch geführt werden und die Datenbank modifiziert werden.

Für die Code-Generierung wird oft lediglich ein Rahmen generiert. Der Grund ist recht einfach:
Programm-Code ist die detaillierteste Form der Systembeschreibung. Es lohnt sich in der Regel nicht, in der Design-Phase alle Details so genau zu beschreiben, wie sie schliesslich implementiert werden. Somit besteht ein Gap in der Detaillierung der Systembeschreibung.

### 4.2.1. Taxonomie

Im einfachsten Fall besteht CASE aus einem Werkzeug, einem Tool für eine spezielle Aufgabe.

Im CASE Umfeld haben sich verschiedene begriffe eingebürgert.

Zunächst unterscheidet man zwischen Upper CASE und lower CASE :
- *Upper CASE* wird in den früheren Phasen der Entwicklung eingesetzt (Analyse, Design)
- *Lower CASE* wird bei der Implementierung, den Tests, Versionskontrollen ... eingesetzt

Wichtiges Hilfsmittel vieler CASE Tools ist der Data Dictionnary, oder in neueren Systemen das Repository (ein wesentlich erweitertes Data Dictionnary, mit Business Rules z.B. für die Datenintegrität). Im Repository oder Data Dictionnary werden die im Projekt verwendeten Datenfelder erfasst, beschrieben und zentral verwaltet. Dadurch wird garantiert, dass Datenfelder in allen Programmen gleich eingesetzt werden und nicht jeder Programmierer sein eigenen Datenfeld für z.B. die Adresse definiert (einmal als Char(20), einmal als Char(45)).

Repositories können auch von vielen Report Generatoren und Screen Painters gelesen werden. Dadurch wird das entwickeln von Berichten und Bildschirmen wesentlich erleichtert.

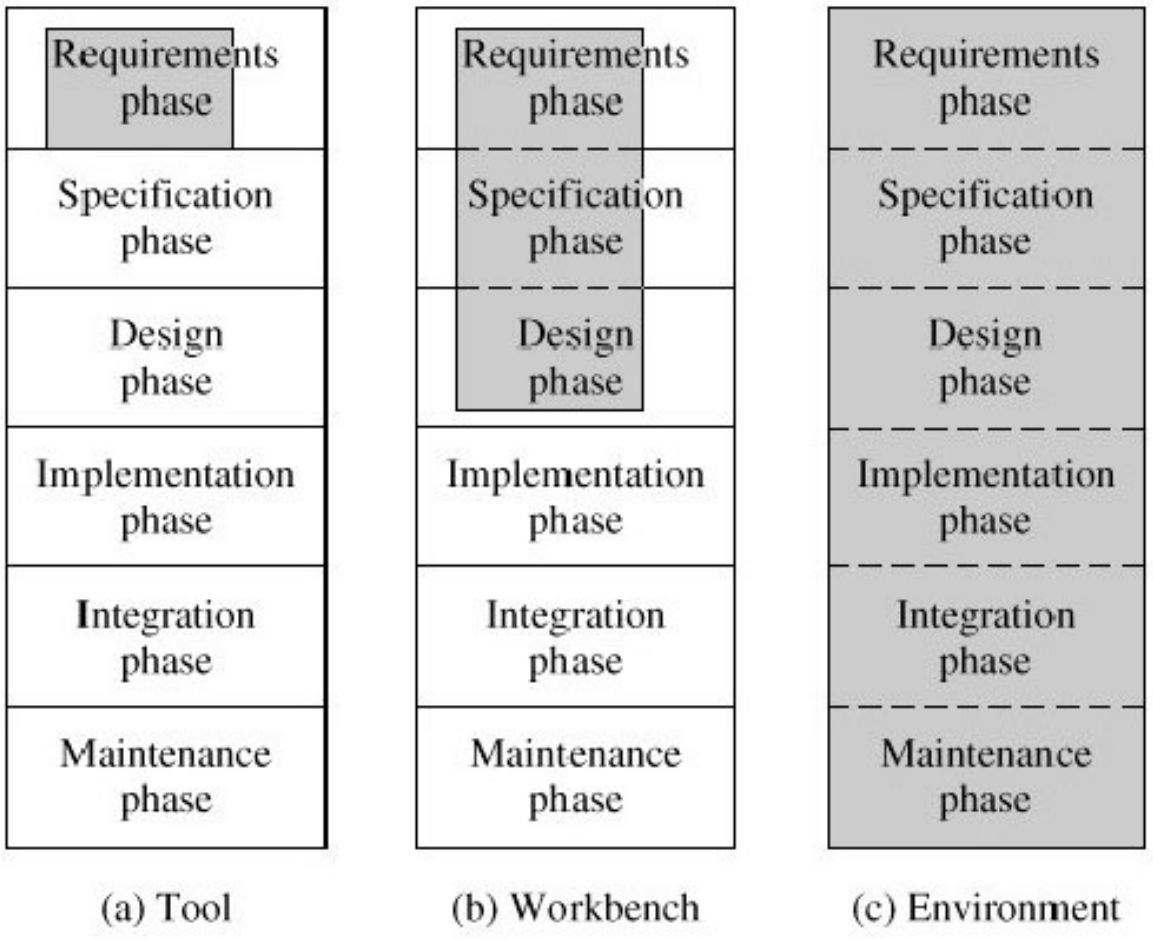
Generatoren sind speziell im Bereich des Rapid Prototyping sehr wertvoll, da damit dem Benutzer eine Idee vermittelt werden kann, wie das ausprogrammierte Programm funktionieren wird. Aus Performance Gründen ist man dann aber oft gezwungen, den Prototypen in einer "tieferen" Programmiersprache zu implementieren, einer Programmiersprache, die effizienter auf der Zielhardware läuft.

Neben UpperCASE und LowerCASE ist ein wichtiger Begriff der des *Workbenches*.

Workbenches kombinieren mehrere Tools, integrieren diese so, dass eine bestimmte Durchgängigkeit erreicht wird. Beispiele dafür sind IEW /ADW, Excellerator und viele mehr.

Wenn versucht wird, den gesamten Lebenszyklus der Software abzudecken, dann spricht man von einem *CASE Environment.* Leider sind die meisten integrierten Tools dadurch gekennzeichnet, dass sie alles nicht gut können, im Gegensatz zu CASETools, die wenigstens eine bestimmte Aufgabe richtig gut können.

#### 4.9.1.1. Zusammenfassung Taxonomie CASE




## *4.10. Scope von CASE*

CASE Tools, im Gegensatz zu Entwicklungs-Umgebungen, sind in der Regel sehr effiziente Hilfsmittel, die dem Entwickler unangenehme Arbeiten abnehmen.

### 4.10.1. CASE und Programmieren im Kleinen

Dem einzelnen Entwickler ist viel geholfen, wenn er folgende Tools (als Auswahl) zur Verfügung hat:

- Source Level Debugger
- Sprachsensitiver Editor
- Struktur Editor (automatische Formattierung beim Eintippen des Programmes)
- Interface Checker (Überprüft die Konsistenz der Schnittstellen)

#### 4.10.1.1. Beispiel: Source Level Debugger

Ein anderes Beispiel kennen Sie bereits : die Microsoft Entwicklungs-Umgebung (Visual Studio, für fast jede Programmier-Sprache, einigermassen einheitlich).

**OVERFLOW ERROR**

Class: CyclotronEnergy

Method: performComputation

Line 6: newValue = (oldValue + tempValue)/tempValue;

oldValue = 3.9583 tempValue = 0.0000

### 4.10.2. CASE und Programmieren im Grossen

Dem Entwicklungsteam helfen andere Werkzeuge, die typischerweise auftretenden Probleme zu lösen:

- Modul-Kontrolle (wer arbeitet an welchem Modul, wer ist dafür verantwortlich)
- Release Kontrolle (Module und Module Versionen, die in ein Release gepackt werden)
- Versions-Kontrolle (Verwalten verschiedener Versionen, inkl dem Generieren von Differenzfiles [was hat sich geändert])
- Projekt-Management Tools (inkl. Erfassung der Projektkosten)

## *4.11. Software Versionen*

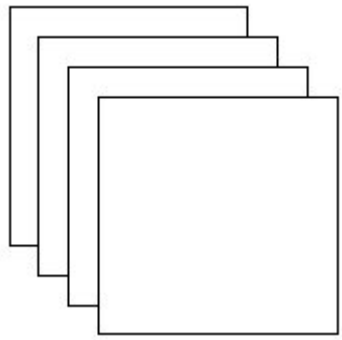
Wesentlich bei Software Produkten, die mit einem Wasserfall , inkrementellen oder Spiralmodell Lebenszyklus entwickelt wurde, ist die Verwaltung der Versionen.

### 4.11.1. Revisions

Annahmen:
Die von Ihnen entwickelte Software ist im praktischen Einsatz, an verschiedenen Orten, rund um den Globus.
Ein Anwender meldet einen schweren Fehler, der unter speziellen Bedingungen auftreten kann.

Ihre Aufgabe ist es, möglichst schnell den Fehler zu beseitigen und alle Anwender, welche die selbe Version der Software einsetzen, zu benachrichtigen und entweder ein Patch (also eine Sofort- Korrektur) oder ein neues Release zu liefern.

Typischerweise werden Sie ab diesem Zeitpunkt verschiedene Revisions, Programmstände, im Gebrauch haben.



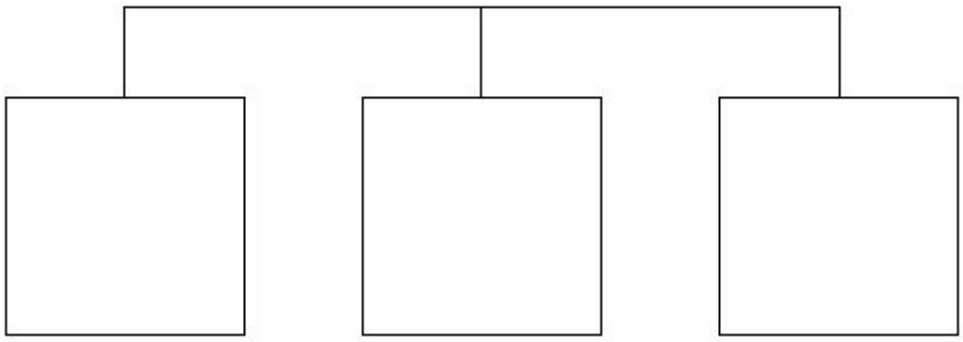
Mehrere unterschiedliche Programmstände, an unterschiedlichen Standorten, bei unterschiedlichen Anwendern im Einsatz.

Die Verwaltung der einzelnen Programmstände geschieht am Besten mit Hilfe einer Versions-Kontroll-Software.

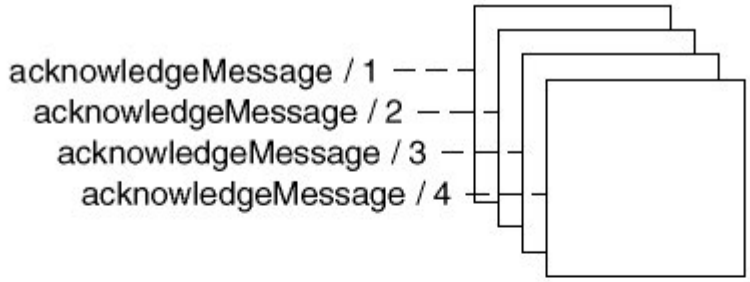
### 4.11.2. Variationen

Neben den Software VERSIONEN gibt es die Software *Variationen.* Als Beispiel betrachten wir eine PC Software, die unterschiedliche Drucker unterstützt.

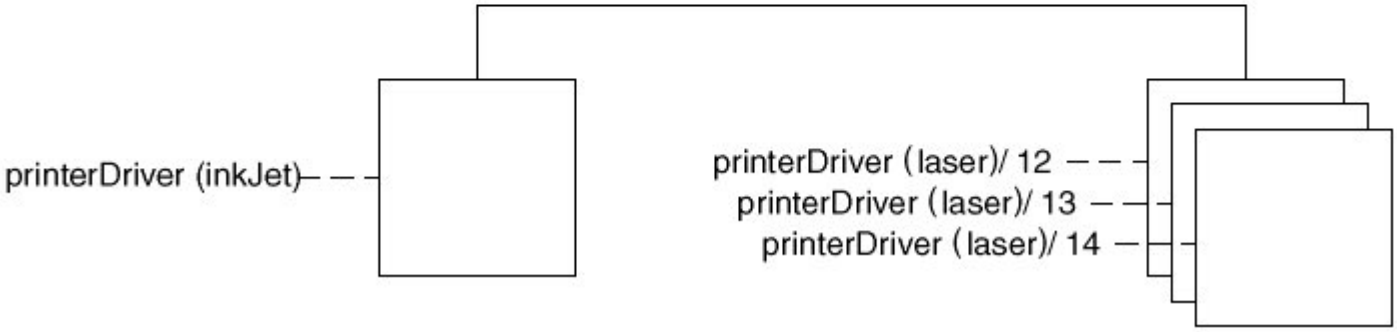
Die Treiber für den Drucker sind zwar immer ähnlich aufgebaut, aber eben nicht identisch. Diese Situation ist nicht vergleichbar mit den Software Versionen!



Variationen existieren auch in unterschiedlichen Versionen. Kombiniert ergibt sich folgendes Bild:




(a)

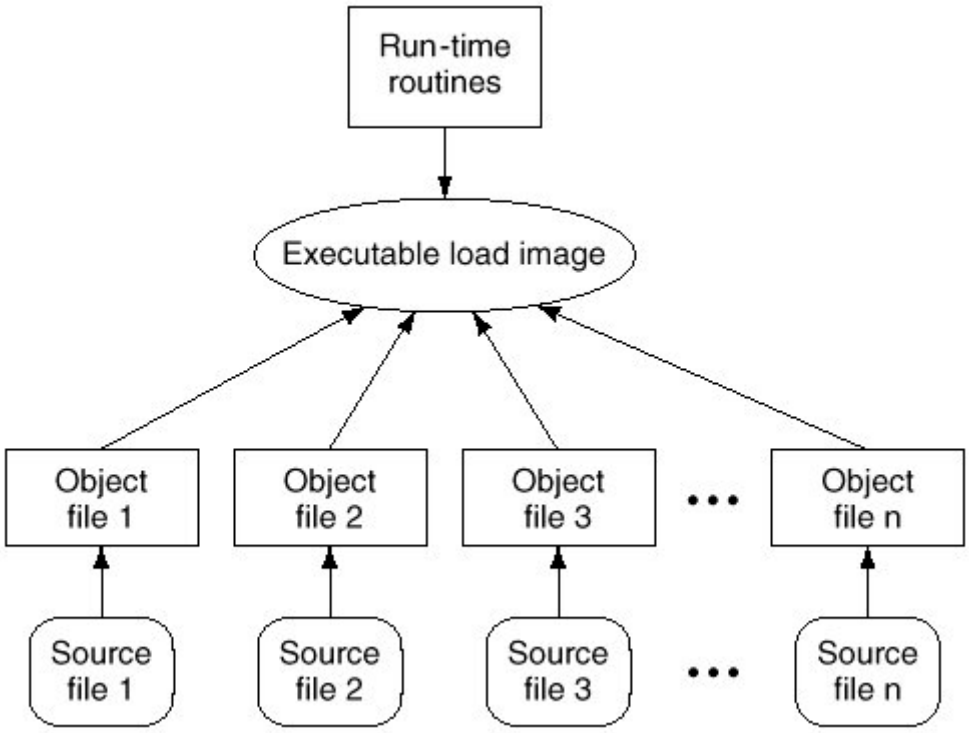



a) zeigt verschiedene Versionen
b)(darunter) zeigt verschiedene Versionen von verschiedenen Varianten

### *4.12. Konfigurations-Kontrolle*

Software besteht (normalerweise) in unterschiedlichen Formen:

- Als Source Files in einer höheren Programmiersprache (C++, ...)
- Als Objekt File , also kombiniert mit den Bibliotheksroutinen
- Als ausführbares Programm




Leider kann jede dieser Formen in unterschiedlichen Versionen und Varianten existieren. Die Wahrscheinlichkeit, dass wir in einem grossen Projekt ein Durcheinander bekommen, falls wir alles von Hand verwalten, ist also sehr gross.

Diese Werkzeuge sind auf fast allen Rechnern bezw. Workstations verfügbar. Auf Grosssystemen sind sie entsprechend mit Bibliotkeksverwaltungs- Programmen kombiniert.

#### 4.12.1. Konfigurationskontrolle in der Wartungsphase

Wenn mehrere Programmierer mit der Wartung von Modulen und Komponenten beschäftigt sind, dann können sehr schnell Konflikte auftreten:

Der Entwickler 1 arbeitet an der Komponente A, welche ihrerseits die Komponente B verwendet;
Der Entwickler 2 arbeitet an der Komponente B, welche also von A benutzt wird.

Falls die Abstimmung der Entwickler untereinander nicht perfekt funktioniert, dann kann es vorkommen, dass der Entwickler 2 Schnittstellen ändert. Dadurch wird der Entwickler 1 sicher grosse Probleme bekommen.

Team- Software erlaubt das Ein- und Aus- Checken von Software - Komponenten und das jeweilige Benachrichtigen der Entwickler über :

- Wer arbeitet an der Komponente?
- Welche Komponenten hängen wie voneinander ab (oft graphisch dargestellt als Baum)?
- Wann wurde welche Komponente zuletzt geändert?

Ohne solche Werkzeuge wäre ein Programmieren im Grossen fast unmöglich.

#### 4.12.2. Baseline

Unter Baseline versteht man die minimale Konfiguration, die nötig ist, um ein Software System produktiv einsetzen zu können.

Der Entwickler lädt, wenn er Änderungen an Komponenten vornehmen will, die benötigten Komponenten in seinen *privaten Arbeitsbereich*. Dort kann er alles austesten und mit den Komponenten machen was er will.

Sobald er die Komponente wieder zurück gibt, wird sie *eingefroren*, bis zur nächsten Änderung.
Die Komponente kann dann in der Baseline, der Grundversion der Software eingesetzt werden.

#### 4.12.3. Konfigurations Kontrolle in der Entwicklungs Phase

Software Komponenten müssen auch während der Entwicklung verwaltet werden. Allerdings geschieht dies nicht oder in der Regel nicht in gleichem Umfang wie in der Wartungsphase, da der Entwickler noch zu viele Änderungen machen muss.

Nachdem eine Komponente aber getestet ist, muss sie einer Versionskontrolle unterstellt werden. Sonst wären Integrationstests fast unmöglich.

### *4.13. Build Werkzeuge*

Unter Unix hat sich ein Werkzeug besonders hervorgetan : MAKE. Auch unter Windows wird beim Einsatz des Developper Studios ein Make File erzeugt.

Aufgabe des Makefiles ist es, vereinfacht gesagt, alle Files, die benötigt werden, um ein ausführbares Programm zu erzeugen, zusammen zu bringen. Das sollte möglichst einfach geschehen, also ohne grossen Aufwand auf Benutzerseite.

MAKE ist schon "uralt", wurde aber bis heute durch nichts ersetzt, was brauchbarer ist.

### 4.13.1. Beispiel eines (N)MAKE Files aus DevStudio (Microsoft) (DSP File)

```
# Microsoft Developer Studio Project File - Name="Arrays" - Package Owner=<4>
# Microsoft Developer Studio Generated Build File, Format Version 5.00
# ** DO NOT EDIT **

# TARGTYPE "Java Virtual Machine Java Project" 0x0809

CFG=Arrays - Java Virtual Machine Debug
!MESSAGE This is not a valid makefile. To build this project using NMAKE,
!MESSAGE use the Export Makefile command and run
!MESSAGE
!MESSAGE NMAKE /f "Arrays.mak".
!MESSAGE
!MESSAGE You can specify a configuration when running NMAKE
!MESSAGE by defining the macro CFG on the command line. For example:
!MESSAGE
!MESSAGE NMAKE /f "Arrays.mak" CFG="Arrays - Java Virtual Machine Debug"
!MESSAGE
!MESSAGE Possible choices for configuration are:
!MESSAGE
!MESSAGE "Arrays - Java Virtual Machine Release" (based on\
 "Java Virtual Machine Java Project")
!MESSAGE "Arrays - Java Virtual Machine Debug" (based on\
 "Java Virtual Machine Java Project")
!MESSAGE

# Begin Project
# PROP Scc_ProjName ""
# PROP Scc_LocalPath ""
JAVA=jvc.exe

!IF  "$(CFG)" == "Arrays - Java Virtual Machine Release"

# PROP BASE Use_MFC 0
# PROP BASE Use_Debug_Libraries 0
# PROP BASE Output_Dir ""
# PROP BASE Intermediate_Dir ""
# PROP BASE Target_Dir ""
# PROP Use_MFC 0
# PROP Use_Debug_Libraries 0
# PROP Output_Dir ""
# PROP Intermediate_Dir ""
# PROP Target_Dir ""
# ADD BASE JAVA /O
# ADD JAVA /O

!ELSEIF  "$(CFG)" == "Arrays - Java Virtual Machine Debug"

# PROP BASE Use_MFC 0
```

```
# PROP BASE Use_Debug_Libraries 1
# PROP BASE Output_Dir ""
# PROP BASE Intermediate_Dir ""
# PROP BASE Target_Dir ""
# PROP Use_MFC 0
# PROP Use_Debug_Libraries 1
# PROP Output_Dir ""
# PROP Intermediate_Dir ""
# PROP Target_Dir ""
# ADD BASE JAVA /g
# ADD JAVA /g

!ENDIF

# Begin Target

# Name "Arrays - Java Virtual Machine Release"
# Name "Arrays - Java Virtual Machine Debug"
# Begin Source File

SOURCE=.\Arrays.java
# End Source File
# End Target
# End Project
```

Es versteht sich von selbst, dass der Unterhalt dieses File recht wichtig ist, aber auch kompliziert.
Im Falle des DevStudio ist jede Änderungen viel aufwendiger als folgendes Vorgehen:

Wann immer Sie Probleme mit dem DSP File haben, beziehungsweise Änderungen in Ihrem Projektfile durchführen möchten, dann ist es einfacher, das File zu löschen und vom DevStudio eine neue Version generieren zu lassen. Das geht viel schneller und ist wesentlich effizienter.

### *4.14. Produktivitäts- Gewinn durch CASE*

Eine Studie (10 Branchen, 45 Firmen) zeigt folgendes Ergebnis:

- Der Produktivitätsgewinn beträgt durchschnittlich 9% - 12%, ist also eher moderat

- Die Software Qualität kann mit Hilfe von CASE verbessert werden
- Die Wartung wird durch Einsatz von CASE vereinfacht

Eine wirtschaftliche Rechtfertigung von CASE alleine mit Hilfe des Produktivität Gewinnes ist NICHT möglich!

### *4.15. Software Metriken*

Ohne Messung des Software Entwicklungs- und Wartungs- Prozesses bleibt auch unklar, wo welche Verbesserungen erzielt werden können.

Dies führte zur Entwicklung / Definition verschiedener *Metriken* (metrics).

Einfache Metriken sind:
- Lines of Code (LOC)
- Fehler pro Komponente (wie messe ich das?)
- Anzahl Personenmonate
- ...

Es gibt heute eine unübersehbare Anzahl Software Metriken und es werden auch immer noch neue erfunden. Das zeigt, dass der Software Entwicklungs- und Wartungs- Prozess sicher noch nicht richtig verstanden wird.

Das Thema ist recht interessant, sprengt aber den Rahmen des Skriptes. Auf den Folien finden Sie detailliertere Angaben (das Skript ist noch in Bearbeitung; zuerst sind immer die Folien [Vorversion des Skriptes]).

### *4.16.*

### *Zusammenfassung*

Miller's Gesetz (7 +/- 2) bedingt eine Zerlegung komplexerer Software Systeme in Module und Komponenten.

Stufenweise Verfeinerung hilft bei der Zerlegung, kann aber auch zu Problemen führen, falls bei einer Zerlegung diese ungeschickt gewählt wurde:
Dies führt zu Folgeproblemen, einer schwierigen Wartung und Unübersichtlichkeit.

CASE Tools werden in einer grossen Zahl angeboten. Unterschiedliche Abdeckungsgrade führen zu unterschiedlichen Bezeichnungen (Tools, Workbench, Environment).

Der Produktivitätsgewinn ist eher dürftig, aber andere Faktoren sind wichtig: Qualität, Wartbarkeit, Personen-Unabhängigkeit.

Metriken wurden für unterschiedliche Entwicklungs- und Wartungs- Modelle entwickelt. Oft fehlt die statistische Basis, um zu testen, ob eine Metrik sinnvoll oder eher sinnlos ist.

### *4.17. Selbsttestaufgaben*

1. Welche Projektorganisation würden Sie wählen, um eine Gehaltsapplikation zu entwickeln.
   Wie würden Sie das Projektteam organisieren, falls Sie Software für einen Überschalljäger entwickeln müssten?
   Erläutern Sie Ihre Antwort!
2. Sie haben gerade Ihr Studium abgeschlossen und starten eine eigene Firma. Alle Ihre Angestellte kennen Sie vom Studium her.
   Könnte eine demokratische Teamorganisation funktionieren?
3. Modifizieren Sie das Masterfile Update Programm :
   Lesen Sie jeweils zwei Update Records und überprüfen Sie die Konsistenz der Daten in folgendem Sinne
   Wenn ein Record zweimal im Update File steht und zweimal INSERT als Operation spezifiziert ist, dann besteht offensichtlich ein Fehler in den Daten.
   Wenn eine UPDATE Operation von einer DELETE Operation gefolgt wird, dann kann dies auch nicht sehr sinnvoll sein, da ja die Änderung nie eingesetzt wird.
4. Warum macht es wenig Sinn, in einer Organisation, die sich auf Level 1 CMM befindet, CASE einzuführen.
5. Welche CASE Tools (konkrete Produkte) möchten Sie für die Entwicklung Ihrer Lösung der CaseStudy Software einsetzen. Klären Sie zuerst ab, welche Software in den Labors oder dem Server vorhanden sind.

6. Lesen Sie folgende Artikel:

- Niklaus Wirth : "Program Development by Stepwise Refinement"
  *Communications of the ACM*, **14** (April 1971) pp. 127-137 [obligatorisch]
- (D. L. Parnas : "A technique for Software Module Specification with Examples"
  *Communications of the ACM*, **15** (May 1972) pp 330-336 )[freiwillig]

  Beide Artikel sind im Buch:
  Writings of the Revolution - Selected Readings on Software Engineering
  Edited by Edward Yourdon
  1982 Yourdon Press ISBN: 0-917072-25-1

*4. **Stepwise Refinement, CASE und andere Werkzeuge…** Fehler! Textmarke nicht definiert.*

**4.1. Stufenweise Verfeinerung ..... 11**
4.1.1. Problemstellung ..... 11

**4.2. Kosten- Nutzen Analyse ..... 15**
4.2.1. Investitionskosten und Betriebskosten ..... 15
4.2.2. Wirtschaftlichkeit ..... 16

**4.3. CASE : Computer Aided Software Engineering ..... 17**
4.3.1. Taxonomie ..... 17

**Scope von CASE ..... 19**
4.4.1. CASE und Programmieren im Kleinen ..... 19
4.4.2. CASE und Programmieren im Grossen ..... 20

**4.5. Software Versionen ..... 20**
4.5.1. Revisions ..... 20
4.5.2. Variationen ..... 21

**4.6. Konfigurations-Kontrolle ..... 23**
4.6.1. Konfigurationskontrolle in der Wartungsphase ..... 23
4.6.2. Baseline ..... 24
4.6.3. Konfigurations Kontrolle in der Entwicklungs Phase ..... 24

**4.7. Build Werkzeuge ..... 24**
4.7.1. Beispiel eines (N)MAKE Files aus DevStudio (Microsoft) ..... 25

**4.8. Produktivitäts- Gewinn durch CASE ..... 26**

**4.9. Software Metriken ..... 27**

**4.10. Zusammenfassung ..... 28**

**4.11. Selbsttestaufgaben ..... 29**