Kapitel 1 – Einführung

Liste von bekannten Fehlern:

- In Warschau rollt ein Airbus über die Landebahn hinaus, weil die Bordcomputer die Auslösung der Schubumkehr verweigert hatten, da nicht alle 3 Fahrwerke den Boden berührt und sich gedreht haben.
- In Denver versagt sie Softwaresteuerung der Gepäcktransportbänder.
- Toll Collect-System geht verspätet an den Start -> Vertragsstrafen
- Zu kurze Kabelbäume beim Bau des A380 durch inkompatible CAD-Versionen

Übliche Ursachen warum Software nicht den Erwartungen entspricht:

- Software ist meistens eingebettet in komplexe Systeme, die nicht primär als Computer dienen.
- Aufgabe der Software in einem komplexen System ist schwer zu definieren
- Das eigentliche programmieren nur ein (kleiner) Teil der Tätigkeit.
- Softwarepakete haben gewaltige Dimensionen und werden von großen Teams hergestellt.
- Digitale Welt impliziert komplexere Abhängigkeiten als die analog-stetige Welt

Die 3 spezifischen Wesensmerkmale der Tätigkeit eines Ingenieurs:

- Kreative Anwendung wissenschaftlicher Grundlagen zur Erbringung einer gewissen Funktion
- Wirtschaftliche Abwägungen (kostengünstigste aller durchsetzbaren Möglichkeiten)
- Sicherheit (Betrieb darf weder System noch Bediener gefährden)

1.3 Warum Software anders ist:

- klassische Ingenieurdisziplinen haben Naturgesetze als Grundlage
  einzig: Es gibt bestimmte Dinge die aus prinzipiellen Erwägungen heraus nicht durch ein Programm beantwortet werden können
- Software ist verschleißfrei, allerdings muss sie in weitaus größeren Zeiträumen geplant werden (siehe Jahr-2000-Problem)
- Bei Hardware kann jedes der Subsysteme erschöpfend getestet werden, dies ist bei Software aufgrund des maßgeblichen Zusammenspiels im System nicht möglich
- Falsche Annahme in der Praxis, dass selbst im letzten Moment noch Aufgaben, Ziele und Umfang noch radikal änderbar wären.

1.3.1 Die Arbeitswelt des Softwareingenieurs:

- Die Anpassung existierender Software kommt häufiger vor als das Schreiben neuer Systeme
- Beim Schreiben neuer Systeme entfällt der größte Marktanteil auf sog. Standard-Software (shrink-warp Software), der größte Arbeitsaufwand auf individuell gefertigte Systeme.

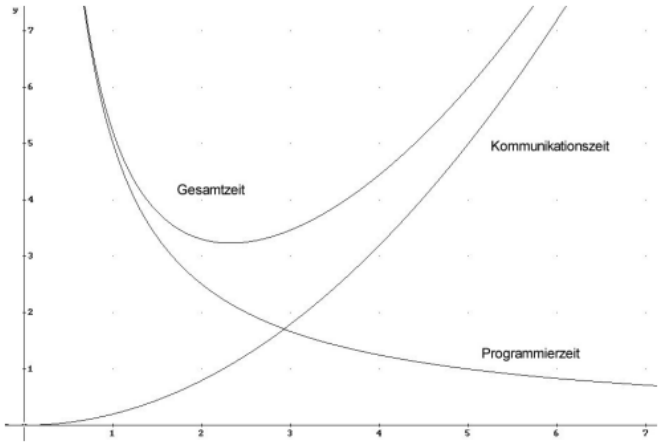
1.4 Programmieren im Großen vs. Programmieren im Kleinen:

- LOC = „Lines of Code“; NCSS = “Non-Commentary Source Statements”
- Mannjahre kennzeichnen den Programmieraufwand (kein Dreisatz anwendbar)

Programme im Kleinen	Programme im Großen
Genaue Spezifikationen liegen vor bzw. werden nicht benötigt	Spezifikationen müssen erst erarbeitet werden
Entwicklung in Einzelarbeit	Entwicklung im Team
Einschrittige Lösung	Lösung in vielen Schritten
Lösung besteht aus einer Komponente	Lösung in viele Komponenten zerlegt
Entwickler und Benutzer einheitlicher Personenkreis	Entwickler und Benutzer ungleiche Vorbildung
Kurzlebige Programme, Unikate	Langlebige Programme, Programmfamilien

1.5 Das Brookssche Gesetz

- Anzahl der Mitarbeiter n, Zeit t Kommunikationsaufwand k
- $t \sim \frac{1}{n}$
- $k \sim \binom{n}{2} \approx c\frac{n^2}{2}$



Kapitel 2 – Modulkonzept

Große Systeme werden in einzelnen Teilen übersetzt.

Modularisierung dient zur

- Bewältigung von Problem- und Programmkomplexität, da Probleme und Aufgaben in Teilprobleme und –aufgaben zerlegt werden.
- Verbesserung der Wartbarkeit, indem Funktionen so lokalisiert werden, dass Änderungen voraussichtlich in nur einem Modul lokal durchgeführt werden können.
- Bewältigung des Arbeitsumfanges, da die Entwicklung verschiedenen Programmteile parallelisiert werden kann
- Meyer’s Enzykl. Lexika: „… ein austauschbares, komplexes Teil eines Geräts oder einer Maschine das eine geschlossene Funktionseinheit bildet.

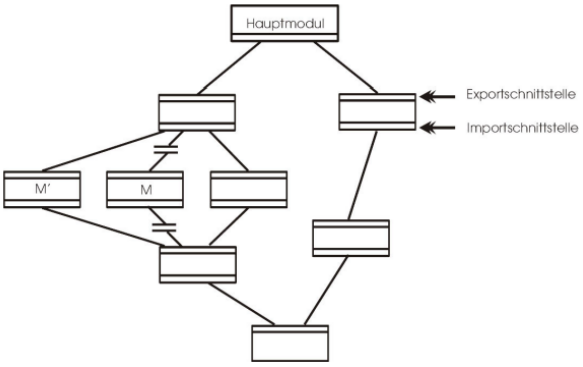
Wesentliche Merkmale:

- Austauschbarkeit ohne Beeinträchtigung des Restsystems
- komplexe Leistung
- geschlossene Funktionseinheit

2.1 Geheimnisprinzipen

Parnas'schen Ideen zum Geheimnisprinzip:

1. Ein Modul kommuniziert mit seiner Umwelt nur über eine klar definierte Schnittstelle
2. Das verwenden von Modulen verlangt keine Kenntnis ihrer inneren Arbeitsweise
3. Die Korrektheit eines Moduls kann man ohne Kenntnis seiner Einbettung in das Gesamtsystem nachprüfen.



Das Gegenteil des Geheimnisprinzips nennt sich „**glasnost-Prinzip**" (verwendet von Microsoft)

2.2 Design by contract

- Idee von Bertrand Meyer
- Es gibt Invarianten, Vor- und Nachbedingungen die Eingehalten werden müssen
- Gegenstück: Defensive Programmierung
- Die Frage des Austauschens eines Moduls M gegen ein Modul M' können wir nun auch präziser diskutieren: M' muss den Vertrag von M einhalten
    - Die Vorbedingungen von M' sind schwächer als die von M: pre(M) ➔ pre(M')
    - Die Nachbedingungen von M' sind stärker: post(M') ➔ post(M)
    - Die Invarianten von M' machen die Invarianten von M nicht ungültig (enthalten keinen Widerspruch zu den Invarianten von M)

2.3 Bestandteile eines Moduls

- Importschnittstelle: Es wird festgelegt, welches Konstrukt von außerhalb verwendet werden.
- Exportschnittstelle: Es wird festgelegt, welches Konstrukt aus diesem Modul von außerhalb verwendet benutzt werden dürfen. (Funktionen, …)
  Meist wird noch weiteres spezifiziert wie: Art der Parameter, Seiteneffekte, Vor-/Nachbed.
- Invarianten: Sind nach außen nicht sichtbar und sagen nur etwas über inneren Zustand aus.
- Rumpf: besteht aus allen Funktionen und ggf. Initialisierungscode
- Oft wird vorgeschlagen funktionale Schnittstellen zu verwenden. (Vorteil: Modul sorgt für Einhaltung der Invariante, Protokollierung möglich)

2.4 Spezifikation von Software

- Trivial: je formaler desto wertvoller die Spezifikation
- Softwaretechnik verlangt die Einhaltung von Normen wie jede andere Ingenieurwissenschaft.
- Der immense Reichtum der mathematischen Formelsprache muss für spezifikationszwecke stark eingeschränkt werden.
- Beispiel JML (Java Modeling Language): requires, ensures, invariant, pure (zur Spezifikation von Methoden ohne Nebenwirkungen), modifies zur Angabe von Nebenwirkungen, \result, \old, \exists, \forall, \sum, \product, …
  Im folgenden Vorteile dazu:
    - Beschränkung auf die Möglichkeiten der Programmiersprache Java
    - single source-Prinzip (Spezifikation steht in Kommentaren)
    - Werkzeuge verfügbar welche die automatische Behandlung solcher Spezifikationen unterstützen, indem sie sich auf der Programmiersprache abstützen.

2.5 Softwaretechniksprachen

Eigenschaften:

- Unterstützung von Softwareabstraktion durch Mechanismen zur
  - Spezifikation einer Schnittstelle
  - Prüfung einer Implementierung gegen die spezifizierte Schnittstelle insbesondere auch auf Vollständigkeit
  - Prüfung des Gebrauchs einer Schnittstelle gemäß ihrer Spezifikation
  - Prüfung des konsistenten Gebrauchs der gleichen Version einer Schnittstelle beim Linken des Programms
- Kapselung von Namensräumen, durch
  - Konstrukt zur Vermeidung von Namenskonflikten (qualifizierte Namen)
  - Gruppierung in Namensräume (Module)

Definitionsmodul: Alle Objekte die aus dem Modul exportiert werden, sind hier definiert.
Implementierungsmodul: Die Implementierung des Moduls

2.7 Keine Softwaretechniksprache? Was nun?

1. Sei möglichst spezifisch in der Notation der Schnittstellen (header files); mache keinen Gebrauch von Erleichterungsmöglichkeiten der Programmiersprache
2. Verwende niemals eine Funktion mit unterschiedlichen Anzahlen von Parametern, auch nicht, wenn die Programmiersprache dies ermöglicht.
3. Prüfe die Vollständigkeit der Implementierung gegen die Schnittstellenbeschreibung: Sind alle dort erwähnten Funktionen implementiert? Haben sie den richtigen Typ? Sind noch Funktionen exportiert, die der Schnittstellenbeschreibung nicht vorkommen.
4. Prüfe die Verwendungsstellen externen Symbole gegen die Schnittstellenbeschreibung: Stimmen die Typen überein?
5. Schaffe getrennte Namensräume, mind. durch die Etablierung und Einhaltung geeigneter Programmiernormen.
6. Schreibe Spezifikationen im Sinne des „design by contract“ in Form von standardisierten Kommentaren (ähnlich JML) in die Funktionsköpfe.

Kapitel 3 – Systementwurf

Definition nach Holbaek-Hanssen: Ein System ist ein Teil der Welt, der von einer Person oder einer Gruppe von Personen während eines bestimmten Zeitraums und zu einem bestimmten Zweck als eine aus Komponenten gebildete Einheit betrachtet wird. Dabei wird jede Komponente durch Merkmale beschrieben, die als relevant ausgewählt werden und durch Aktionen, die in Beziehung zu diesen Merkmalen und zu den Merkmalen anderer Komponenten stehen.

Folgerungen:

- Jedes System verfügt über *Schnittstellen* zum Rest der Welt.
  Schnittstellen != Modulschnittstellen
- System hat inneren *Zustand* (Geheimnisprinzip)
- Komponenten eines Systems können *Aktionen* ausführen (Methoden) → Zustandsänderung
- System zeigt gewisses *Verhalten* wie es auf Folgen von Eingaben reagiert.

Vorher gehende Phasen:

- **Problemanalyse**: Analyse wird nach rein logischen Gesichtspunkten durchgeführt, um die Verständlichkeit zu fördern.
- **Analysephase**: Es wird das zu entwickelnde System also aus der Sicht des Benutzers beschrieben.
- **Entwurfsphase**: Es wird das zu entwickelnde System aus der Sicht der Maschine beschrieben. (üblich Bildung abstrakter Maschinen, welche stufenweise Fähigkeiten abstrahieren)

3.2 Systemarchitektur

Beschreibungsansätze (weniger echte Vorgehensmodelle):

- **top down**: Geht von der höchsten Hierarchiestufe aus und verfeinert diese Stück für Stück
- **bottom up**: Geht von der logisch niedrigsten Hierarchiestufe aus und beschreiben zuerst die Bausteine, die dann schrittweise zu komplexeren Einheiten zusammengesetzt werden.

Der Forscher soll nur dasjenige als wahr annehmen, was der Vernunft so klar ist, dass jeglicher Zweifel ausgeschlossen ist, größere Probleme in kleinere aufspalten, immer vom Einfachen zum zusammengesetzten hin argumentieren und das Werk einer abschließenden Prüfung unterwerfen.

Kernfragen beim Systemenwurf:

1. Wie sehen die Eingaben für das System aus?
2. Welche Ausgaben soll das System erzeugen?
3. Welche Datenstrukturen werden als Grundlage eines Algorithmus benötigt, um Eingabe und Resultate im Programm abzulegen
4. Welche Operationen lassen sich auf diesen Strukturen durchführen?
5. Welche davon sind weitgehend unabhängig voneinander? (d.h. für welche müssen keine gemeinsamen Invarianten unterhalten werden?)

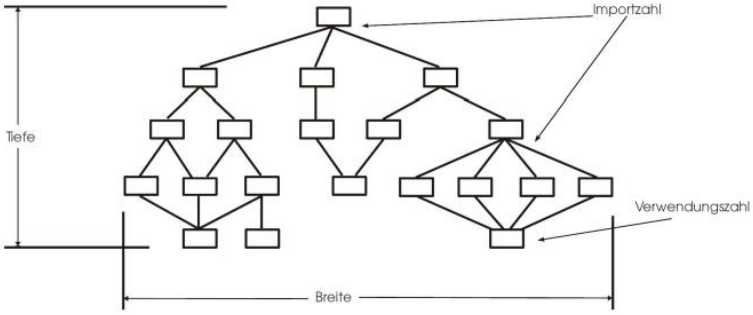
Modulentwurf muss datenorientiert vorgehen.

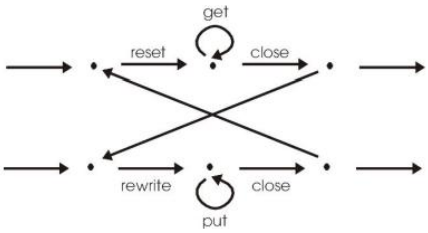
Entwurfsempfehlungen:

1. Geheimnisprinzip und design by contract
2. Entwurfsentscheidung soll nur in einem Modul verkörpert sein.
3. Jedes Modul sollte mind. eine Entwurfsentscheidung beinhalten.
4. Entwurfsentscheidung soll mit ihrer Begründung sorgfältig dokumentiert werden.
5. Schnittstellen sollen möglichst klein sein, sowohl auf Funktions- als auch auf Modulebene.
6. Modulstruktur sollte hierarchisch sein und nicht zyklisch (Zyklus nur innerhalb eines Moduls zuslässig)

3.5 Qualität eines Entwurfs

1. Minimalität der Schnittstellen
2. Ausgewogenheit gewisser Maße:
    - *Modulgröße* ist umgekehrt proportional zur *Modulzahl*
    - Importzahl (fan out)





- Verwendungszahl (fan in)
- Tiefe/Breite der Modulstruktur

3. Modulbinding („cohesion“): Eigenschaft eines Moduls
   - zufällige Bindung
   - logische Bindung
   - temporale Bindung (z.B. nur Initialisierungsfunktionen)
   - prozedurale Bindung: Elemente müssen in einer bestimmten Reihenfolge ausgeführt werden (z.B. File I/O)
   - Kommunikationsbindung
   - Sequentielle Bindung
   - Funktionale Bindung: Es wird nur eine einzige Funktion exportiert.
4. Modulkopplung („coupling“): Eigenschaft zwischen zwei Modulen
   - Datenkopplung, ADT-Kopplung, Steuerkopplung, Externe Kopplung, Inhaltskopplung

3.6 API-Entwurf

Hennings Hinweise für die Entwicklung von APIs:

- Eine API sollte möglichst klein sein.
- APIs können nur entworfen werden, wenn ihr Einsatzkontext verstanden ist.
  (z.B. null oder Exception falls kein Wert in der HashMap gefunden?)
- APIs sollten aus der Perspektive des Aufrufers geschrieben werden.
- APIs sollten dokumentiert werden, bevor sie implementiert werden.

Kapitel 4 – Objektkonzept, OOD und OOA

Probleme mit zweitklassigen OO-Sprachen wie C++ und Java:

- Die dynamische Erzeugung und Vernichtung von Objekten wird vermischt mit Fragen der Speicherallokation und -deallokation. -> garbage collector
- Zeiger anstatt URI (Uniform Ressource Identifier). Somit sind keine persistenten Objekte möglich
- Datentypen und Klassen werden miteinander vermischt.

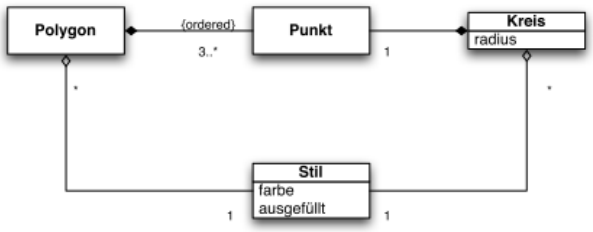
4.2 Statische Modellierung mit Unified Modeling Language (UML)

Bedeutungen des Begriffs Klasse:

- Konzept: In der Analysephase ist eine Klasse meist nicht mehr als ein bestimmtes Konzept
- Typ: Klassen werden oft im Sinne von (Datentypen) verwendet.
- Objektmenge: Werden als Menge gleichartiger Objekte betrachtet
- Implementierung: Eine Klasse bezeichnet dann auch meist noch den Programmtext einer Klasse (z.B. in Java)

4.2.1 Assoziation

- Allgemeiner Typ von Beziehungen (Relationen) zwischen Klassen. (eventl. mit Kardinalität)
- Mit geschweiften Klammern können Einschränkungen hinzugefügt werden



4.2.2 Aggregation & Komposition

- eine "has-a-relation" oder "part-of-relation" nennen wir Aggregation (weißer Raute)
- besonders starke Formen nennen wir Komposition (schwarze Raute)

4.2.3 Vererbung

- Vererbung wird im wesentlichen mit zwei zielen eingesetzt:
  - Zur Formalisierung einer can-be-used-as-Beziehung
  - Zur Wiederverwertung von Implementationsteilen.
- **statische Bindung**: Der Compiler ist in der Lage, diese Auszahl zu treffen.
- **dynamische Bindung**: Erst zur Laufzeit steht der Typ fest
- pragmatische Lösung für Mehrfachvererbung sind Interfaces (Java)

4.3 Dynamische Modellierung

- Hier geht es um die Beschreibung von Vorgängen im System, die einen gewissen Zeitverlauf und Zustandsveränderungen haben
- Zustandsübergangsdiagramme (Transitionsdiagramme): im Prinzip endliche Automaten
- Sequenzdiagramme (hier geht es um tatsächliche Objekte)

4.4 Model driven architecture

Ein Systemmodell in vier getrennten Schichten:

- **Computation Independent Model** (CIM): Hier werden die Geschäftsprozesse auf hohem Abstraktionsniveau beschrieben. Keine Rücksicht darauf wann Programme zum Einsatz kommen.
- **Platform Independent Model** (PIM): Hier geht es um die Geschäftsmodellanteile, die von Computern übernommen werden. Auch noch rein problemorientiert, sie nehmen keine Details irgendeiner Lösung vorweg. (Attribute werden als öffentlich betrachtet)
- **Platform Specific Model** (PSM): Ab hier geht es um die Lösung, daher liegt ein kreativer Schritt im Übergang vom PIM zum PSM ("low-level design"). Hier werden viele Entwurfsentscheidungen getroffen. (um Invarianten einer Klasse zu garantieren müssen Attribute privat sein)
- **Code model**: Dieses Model entspricht bereits der Implementierung in ausprogrammierter Form.

4.5 Prozess einer objektorientierten Modellierung

4.5.1 Definition der Klassen

- Verb-Substantiv-Methode bietet geeigneten Ansatz für die Definition
  Substantive stehen für Klassen und ihre Attribute (Klassen stets im Singular)
  Verben stehen für Methoden und Assoziationen
- Klassen welche Namen physikalischer Gegenstände tragen stehen meist nur für Datensätze.
- Auf der Suche nach relevanten Klassen konzentriert man sich am besten auf:
  - Strukturen, organisatorische Einheiten
  - Rollen, die Personen übernehmen können
  - Andere Systeme, mit denen das zu entwickelnde System zusammenarbeitet
  - Ereignisse, die gespeichert werden müssen
  - Orte

4.5.2 Einführung der Struktur

- Zuerst nach Assoziationsstrukturen suchen, dann Klassifikationsstrukturen
- Klassifikationsstrukturen findet man durch folgende Punkte und Fragen:
  - Ist die aufgefundene Menge von Attributen/Methoden für jedes Vorkommen der Klasse relevant?
  - Ist die betrachtete Klasse ein Spezialfall einer anderen? Haben Klassen gemeinsame Attribute/Methoden?
  - Oberklasse hat nur eine Berechtigung, falls es Situationen gibt in denen mehrerer Unterklassen gleichartig behandelt werden.
  - Unterklasse muss den Kontrakt ("design by contract") der Oberklasse einhalten.
- Nichts ist so starr wie eine einmal eingeführte Klassenhierarchie

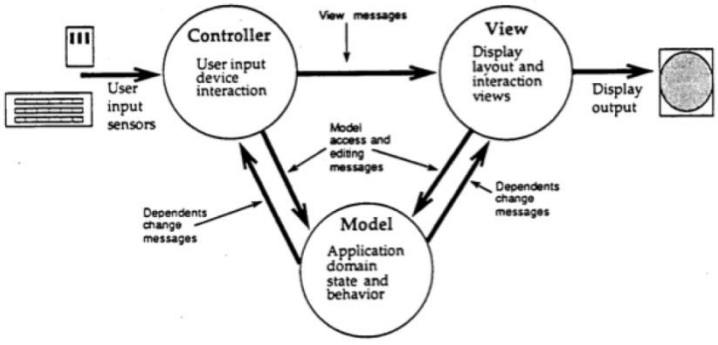
4.5.3 Aufteilung des Modells in Subsysteme

- Subsysteme so definieren, dass höchstens 7 Klassen oder Subsysteme in einem Subsystem zusammengefasst werden.

4.5.4 & 4.5.5 Definition der Attribute & Methoden

- Attribute & Methoden sind so hoch wie möglich in der Klassifikationsstruktur anzusiedeln.
- Methoden erst nach der statischen Modellierung suchen. Faustregel: "Die Daten zuerst"

4.6 Muster



- Erstes OO-Muster "model-view-controller": Beschreibt einen Entwurf zur Entwicklung von graphisch interaktiven Systemen.
  - Model: Zentrale Zustandsdaten, alle Methoden die zur Bechreibung der Zustandsänderungen des Modells notwendig sind
  - View: zeigt Teilmenge der Informationen des Modells.
  - Controller: Interpretiert die graphischen Eingaben und passt die Ansicht dementsprechend an und sendet möglicherweise Änderungsanforderungen zum Model.

4.6.1 Dokumentation objektorientierter Muster

- Es gibt eine von Gamma vorgeführtes Muster zur Beschreibung von Mustern:
  - Name und Klassifikation des Musters
  - Absicht
  - Alias (Abkürzungen)
  - Motivation
  - Anwendbarkeit
  - Struktur (Struktur der am Muster beteiligten Klassen)
  - Teilnehmer (enthält Klassen und Objekte, die an dem Muster beteiligt sind)
  - Zusammenwirken
  - Konsequenzen
  - Implementierung
  - Beispielcode
  - Bekannte Anwendungen
  - Verwandte Muster

4,8 Verteilte objektorientierte Anwendungen

- Im OO-Sektor hat sich das client-server-Paradigma sehr etabliert.
- Weiterdenken führt zu einer benötigten Vermittlungsfunktion, da verschiedene Betriebssysteme mit verschiedenen Programmiersprachen miteinander kommunizieren müssen

4.9 Komponenten und Frameworks

- Frameworks sind wiederverwendbare, sozusagen halbfertige Anwendungen, die sich leicht zu vollständigen Programmen konfigurieren lassen.
- Frameworks werden in 3 unterschiedliche Kategorien eingeteilt:
  1. **System Infrastructure Frameworks**: Dient zur Entwicklung der Infrastruktur von Systemen (z.B. kommunikations-Frameworks)
  2. **Middleware Integration Frameworks**: Dienen dazu Modulare wiederverwendbare Software zu entwickeln.
  3. **Enterprise Application Framework**: Unterstützt die Entwicklung von Applikationssoftware, die direkt von einem Endanwender genutzt wird.
- Typisch für alle Frameworks ist das sogenannte Hollywood-Prinzip der "Call-backs".
  Modulbibliothek bietet Funktionen an
  Framework bietet Schnittstellen um Funktionen anzumelden die dann aufgerufen werden wenn das Framework das entscheidet.

Kapitel 5 – Softwarequalität

Gabriel listet unter anderem die folgenden Eigenschaften von Software auf, welche die "**Qualität ohne Namen**" besitzt:

- kleinen Teil davon anschaue, kann ich sehen, was vor sich geht
  (die Abstraktionen müssen für sich alleine Sinn ergeben)
- großen Teil im Überblick anschaue, kann ich sehen was vor sich geht (nicht alle Details)
- jedes Detail genau so durchdacht, wie jede andere Ebene
- Jeder Teil des Code ist transparent und klar.
- Alles daran kommt mir vertraut vor.

Boehm und Basili listen die **10 wichtigsten Erkenntnisse zur Reduzierung von Fehlern** in Software wie folgt auf:

1. Ein Softwareproblem nach der Auslieferung zu finden und zu beheben, ist oft 100mal teurer, als es während der Anforderungs- und Entwurfsphase zu finden und zu beheben wäre.
2. Heutige Softwareprojekte verwenden ca. 50% ihres Aufwands auf vermeidbare Überarbeitungen (zum Beispiel durch bessere Analyse des Problems)
3. ca. 80% der vermeidbaren Überarbeitungen gehen auf 20% der Fehler zurück.
4. ca. 80% der Fehler stammen aus 20% der Module (ca. 50% fehlerfrei)
5. ca. 90% der Ausfallzeiten geht auf 10% der Fehler zurück (bei IBM sogar nur 0,3%)
6. Inspektionen durch Kollegen finden 60% der Fehler
7. Perspektivenbasierte Inspektionen finden 35% der Fehler als ungerichtete Inspektionen.
8. Disziplinierte persönliche Praktiken können die Fehler-Einführungsraten um bis zu 75% senken.
9. Es kostet ca. 50% mehr pro Quellinstruktion hochzuverlässige Produkte zu entwickeln als Produkte mit niedriger Zuverlässigkeit.
10. 40-50% der Benutzerprogramme enthalten nicht-triviale Fehler. (hier geht es um Fehler die der Benutzer selbst verursacht)

Irrmeinungen zum Testen (Graham):

1. Falsch: Anforderungen stehen am Anfang, Tests am Ende.
2. Falsch: Testen kann man erst, wenn das System fertig ist.
3. Falsch: Anforderungen braucht man zum Testen, aber nicht umgekehrt.
4. Falsch: Wenn es schwierig ist, Testfälle zu finden, dann ist das nur ein Problem des Testens.
5. Falsch: Kleinere Änderungen in den Anforderungen haben wenig Auswirkungen auf das Projekt.
6. Falsch: Zum Testen braucht man die Anforderungsdefinition nicht wirklich.
7. Falsch: Testen kann man nicht ohne Anforderungsdokument.

Etwa 2/3 der Fehler treten so selten auf, dass sie in der Regel kaum als solche wahrgenommen werden. Nur ca. 2,2% der Fehler treten häufig auf, so dass die Korrektur wirklich einen direkt spürbaren Effekt liefert.

5.2 Fehlerdichte

- wirklich gute Software hat ca. 2-3 Fehler pro 1000 Zeilen (durchschnitt bei 25 pro 1000 Zeilen)
- Erfahrene Programmierer machen nicht weniger Fehler und ihre Fehler sind meist wesentlich schwieriger zu entdecken, da sie sehr komplex und subtil sind.
- Ein angenommener weise fehlerfreies Softwareprodukt wird durch eine typische Anzahl typischer Fehler manipuliert. Nach dem Durchlaufen des Qualitätszyklus kann dann eine

statistische Aussage getroffen werden, wie gut die Qualitätssicherung ist. (error seeding)
-> Fehler sind nicht gleichverteilt -> nicht so aussagekräftige Resultate

- Aus der Erfahrung kann ein Entwickler die Fehlerdichte eines alten Produkts als Erwartungswert für das neue ähnliche Produkt ableiten.
- Die wesentliche Bestimmungsvariable dafür wo wahrscheinlich viele Fehler auftreten ist die Anzahl der Zeilen (LOC).

5.4 Dynamische Analyse: Testen

- Ein Testfall gilt als erfolgreich, wenn es gelungen ist, mit seiner Hilfe einen Fehler nachzuweisen.
- Inspektionsmethoden sind im Ergebnis erfolgreicher als Testfälle.
- Hauptkategorien: black box test &white box test

5.4.1 Ablaufbezogenes Testen (structured testing)

- Ein Testfall wird mit der Absicht konstruiert bestimmte Abläufe im Programm zu provozieren.
- Es geht darum möglichst viele Pfade durch das Programm zu erzwingen (Pfadüberdeckung)
- Es lassen sich nur bestimmte Arten von Fehlern entdecken, dazu gehören: unerreichbarer Code, endlose Schleifen, unvollständige/widersprüchliche Bedingungen
- Niemals gefunden werden: vergessene Funktionen, unberücksichtigte Daten, inkonsistente Verwendung von Schnittstellen, ...

5.4.2 Datenbezogenes Testen (volume testing, stress test)

- Testfälle werden auf Basis der Datenbeschreibungen konstruiert.
- Meist auf Basis von Wahrscheinlichkeitsverteilungen werden gültige Kombinationen von Eingabedaten zufallsmäßig erzeugt. (Ergebnis im Prinzip unvorhersehbar)

5.4.3 Funktionsbezogenes Testen (unit tests)

- Die Spezifikation gilt als Basis für die Ermittlung der Testfälle.
- Im Grund die sinnvollste Art der Softwaretests.

5.4.4 Back-to-back-testen

- Hier werden mehrere Versionen des gleichen Programms bezüglich ihrer Funktionalität miteinander verglichen.
- Eine wichtige Variante hiervon ist der Regressionstest.

5.5 Statische Analyse

5.5.1 Vollständigkeit und Konsistenz

- Bei sicheren Programmiersprachen ist eine erfolgreiche Kompilation Indiz für das Fehlen von Unvollständigkeiten und Inkonsistenzen.
  Ermöglichen trotzdem ungenutzte Variable oder nicht erreichbarer Quellcode.
- Bei C/C++ wird dies mit lint weitestgehend auch erreicht.
- **Kiviat-Diagramm**: Eine Auswertung eines Programms nach mehreren Metriken

5.5.2 Inspektionen

- Geschieht meist im Rahmen von formell veranstalteten Besprechungen
- Begriffe: code review, structured walkthrough, formal technical review
- *Beispiel IBM (Fagan)*: Vergleicht ein sogenanntes I-Dokument mit einem S-Dokument (entdeckt fast 2/3 der Fehler, 2,2 Mannstunden pro Fehler notwendig)
  Eingangskriterien: Wenn je nach Art des zu prüfenden Dokuments nicht alle Vorbedingungen erfüllt sind kann die Inspektion nicht beginnen.
  Sitzung: Mit Softwareentwickler, Softwaretester, Moderator, Protokollant
  Ausgangskriterien: Legen fest, wann ein einzelner Inspektionsprozess als beendet gelten kann
- **Klassifikation von Fehlern**:
  - nach *Typ* (unvollständig, unkorrekt, überflüssig in Bezug zur Aufgabe)
  - nach *schwere* (formale Fehler, funktionale Fehler, offene Fragen)
  - nach *Kategorien* (Datendefinition, Registerbenutzung, Logik, Wartbarkeit, Normen, ...)
  - nach *Fehlerquelle* (welcher Entwicklungsschritt hat den Fehler verursacht?)

5.6 Effizienz der Fehlerbeseitigung

- Verfahren die nicht mindestens 85% Effizienz in der Fehlerbeseitigung bringen gelten als unakzeptabel, da solche Programme vom Benutzer als unzuverlässig betrachtet werden.
- Effizienz der 4 Techniken:
  - Formelle Inspektion des Entwurfs (Design Inspection)
  - Formelle Inspektion des Programmcodes (Code Inspection)
  - Formelle Qualitätssicherungsverfahren (Quality Assurance)
  - Formelles Testen der Software (Formal Testing)
- Kombination aus DI und CI tragen bereits den größten Teil zur Qualität bei.

5.7 Value-driven testing

- ca. ein Testfall je 20 Zeilen OO-Code sind notwendig
- Formel für den Testaufwand nach Sneed und Jungmayr (2011):

$$A = AF * \left[\left(\frac{TF * TU}{TP + TP * (1 - TA)} \right)^{TE} * \frac{MT}{TB} \right]$$

  A=Testaufwand, AF=Korrekturfaktor (1), TF=Anzahl der Testfälle, TU=Testüberdeckung, TP=manuelle Testproduktivität, TA=Testautomatisierungsgrad, TE=Skalierungsexponent (1,03), MT=mittlere Testbarkeit, TB=Testbarkeitsmetrik (0,45)

5.8 Verifizieren

- Verifizieren ist keine Alternative zum Testen, sondern vielmehr ein anderer Aspekt des Problems.
- In der Regel ist keine Formale Spezifikation vorhanden, daher ist das Verifizieren nicht möglich.
- Ein Experte kann zwischen 1000 und 2000 Zeilen Quellcode im Jahr verifizieren

Kapitel 6 – Vorgehensmodelle für die Softwareentwicklung

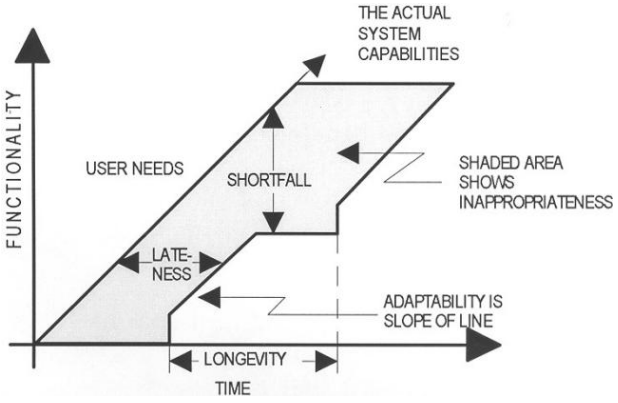
6.1 Software-Lebensläufe

- Unterscheidung zwischen dem Konzept System (bestehend aus Hard- und Software) und Software.
- *3 Grundlegende Probleme*:
  - **Abgrenzungsproblem**: Phasen sind meist fließend im Übergang, speziell bei großen Projekten kann manches sich noch in der Planung befinden wobei andere Teile schon in Gebrauch sind
  - **Abfolgeproblem**: Einzelne Phasen laufen nie streng nacheinander ab, es kommt häufig zu Rückschritte durch z.B. Planungsfehler
  - **Angemessenheitsproblem**: Ein allgemeines Schema, das auf wirklich alle Projekte passt, hat fast keinen Informationsgehalt mehr. Zu detailliert passt es allerdings auf kaum Projekte mehr.
- Vorgehensmodelle sollen nicht allzu pedantisch ernst genommen werden.

Produktivitätsmetrik

- **Defizit**: Ein Defizit ist ein maß dafür, wie weit die Funktionlität eines Systems zu einem Zeitpunkt t hinter den Anforderungen zurückbleibt.
- **Verspätung**: Ein Maß für die Zeit zwischen dem erkennen einer neuen Anforderung und ihrer Implementierung im System.
- **Änderbarkeit**: Ein Maß für die Geschwindigkeit, mit der ein System an neue Anforderungen angepasst werden kann (Steigung der Funktionalitätslinie)
- **Langlebigkeit**: Die Zeit die das System anpassungsfähig ist, das heißt also den Zeitraum von der ersten Entwicklung über die Wartung bis zur Ablösung durch ein neu entwickeltes System.
- **Unangemessenheit**: Beschreibt den zeitlichen Verlaufs des Defizits (schattierte Fläche)

6.2 Kritik am Wasserfallmodell



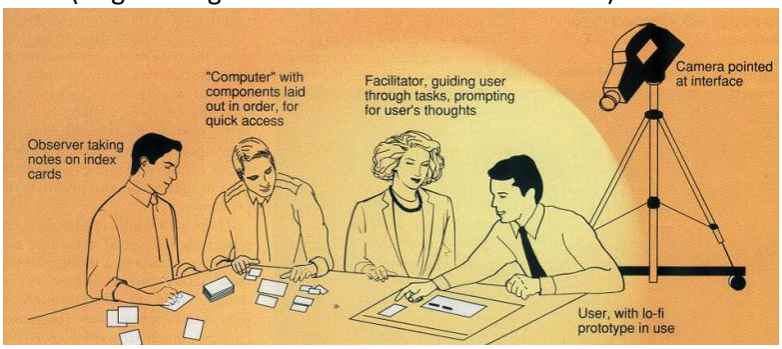
1. Dieses Modell eignet sich nur für relativ einfach Projekte, bei denen wir tatsächlich in einem Wurf Erfolg haben können.
2. Frühes festschreiben der Anforderungen ist problematisch, da Änderungen dann teurer werden
3. Einführung des Systems sehr spät, dadurch später ROI (return on investment)
4. Einführung auf einen Schlag (big bang). Dadurch treten Probleme mit der Handhabung auf, die meist auf Softwareprobleme umgemünzt werden.

6.4 Frühe Prototypen (rapid prototyping)

- es geht darum möglichst schnell einen oder mehrere Prototypen eines Systems zu gewinnen, um damit zu experimentieren und Erfahrungen zu sammeln
- Unterscheidung zwischen
  - **explorativen Prototypen**: Zur Ermittlung von bestimmten Vorstellungen der Benutzer. ("throwaway prototypes")

- **evolutionären Prototypen**: Es werden jeweils verbesserte Versionen des Systems hergestellt, die zunächst prototypisch einsetzbar sind, aber zu weiten Teilen ihren Eingang in das fertige Produkt finden. (Abgrenzung zu inkrementellen Bau schwer!!)

- "lo-fi prototypes": Mit Stift & Papier eine GUI simulieren.



6.5 Inkrementeller Bau

Vorteile:

- Rückmeldung der Benutzer während des Betriebs der ersten Systemstufen
- Vermeidet durch Stufenweise Einführung den Big-Bang-Effekt
- Der Kapitlarückfluss beginnt früher, da erste Systemstufen bereits nach kürzerer Zeit verfügbar sind.
- Gesamtsystem wird früher als gewöhnlich fertig, weil durch Rückkopplung mit dem Benutzer früh vermeintlich wichtige Teile als unwesentlich erkannt werden.

Nachteile:

- Die Reibungslose Integration der Inkremente hängt entscheiden von der gewählten Architektur ab. Bei ändernden Anforderungen im Verlauf der Entwicklung müssen schon fertiggestellte Teile neu entwickelt werden. -> tendiert Richtung Wasserfallmodell
- Die mehrmalige Einführung von Inkrementen erhöht den organisatorischen Aufwand und kann Endbenutzer verwirren.
- Üblicher weiße muss das erste System auch verworfen werden und ist somit wieder ähnlich zu den frühen Prototypen.

6.6 Agile Methoden

- Bezeichnet die Methoden die auf hohe Interaktion mit dem zukünftigen Benutzer Wert legen und andererseits den stetigen Wandel der eigentlichen Projektziele in den Vordergrund rücken.

6.6.1 Extreme Programming (XP)

- Auftraggeber definiert Funktionalitäten des Systems in Form von User Stories und liefert gleichzeitig Testfälle, anhand deren man die Implementierung dieser Szenarien prüfen kann.
- Programmierer geben für User Stories Abschätzungen ab wie lange man dafür benötigt. (User Story darf nie über 3 Wochen dauern!)
- Jeder kann jedes Stück Code verändern
- Auf genaue Spezifikationen und Dokumentationen wird meist verzichtet.
- Ähnelt der Hackerkultur.
- Es wird nur zu zweit an einem Rechner entwickelt (pair programming)
- Die 12 Kernpraktiken (in Kategorien):
  - Feingranulare Rückkopplung:
    - TDD - **TestDrivenDevelopment**: Testgetriebene Entwicklung mit Unit Tests, customer tests (Akzeptanztests (vom AG vorformulierte Testfälle))
    - PG - **PlanningGame**: Planungsspiel
    - WT - **WholeTeam**: Ganzheitliches Mannschaft
    - PP - **PairProgramming**: Programmieren in Paaren

- Stetiger Prozess statt schubweise Bearbeitung
  - CI - **ContinousIntegration**: Stetige Integration
  - DI - **DesignImprovement**: Entwurfsverbesserung
  - SR - **SmallReleases**: Kleine Releases
- Gemeinsames Verständnis:
  - SD - **SimpleDesign**: Einfacher Entwurf
  - SM - **SystemMethaphor**: System-metapher
  - CCO - **CollectiveCodeOwnership**: Gemeinsames Eigentum am Code
  - CS - **CodingStandard**: Programmiernorm oder Programmierkonventionen
- Wohlergehen der Programmierer
  - SP - **SustainablePace**: Nachhaltige Geschwindigkeit

Rechte des Softwareentwicklers:

- klare und fortgesetzte Kommunikation mit dem Kunden zu haben
- jederzeit Unterstützung für seine Qualitätsarbeit zu finden
- die Verantwortlichkeit für deine Tagespläne und -ziele selbst zu haben
- Hilfe von Kollegen zu erbitten und zu bekommen
- eigene Abschätzungen zu machen und anzupassen
- ein nachhaltiges Arbeitstempo
- eigene Entwicklungswerkzeuge, wo immer passend einzusetzen

Pflichten des Softwareentwicklers:

- zu verstehen was benötigt wird und warum es die Priorität des Auftraggebers ist
- Qualitätsarbeit abzuliefern
- andere zu betreuen und alle Fertigkeiten zu teilen
- deine Abschätzungen so genau wie möglich zu machen
- die Konsequenzen deiner Aktionen zu tragen

Rechte des Auftraggebers:

- die Priorität für jede User Story für dein Unternehmen festzulegen
- einen überblicksplan zu bekommen und zu wissen was erreicht werden kann, wann und zu welchen Kosten.
- aus jeder Programmierwoche den maximalen Gegenwert zu erhalten
- Fortschritt im laufenden System zu sehen
- Entwickler über Änderungen in den Anforderungen zu beraten
- jederzeit das Projekt zu beenden und ein nützliches System zu behalten

6.6.2 Refactoring

- Eine Änderung der internen Struktur von Software mit dem Ziel, sie leichter verständlich oder billiger modifizierbar zu machen, ohne dabei ihr beobachtbares Verhalten zu ändern.
- **bad smells** sind zum Beispiel: Duplicated code, Long method, Large class, Shotgun surgery (ein Konzept auf zu viele Klassen verteilt), feature envy (falls eine Methode mehr auf andere Klassen zugreift als auf eigene Daten)

6.6.3 Scrum

- Zwei wesentliche Rollen:
  - **Product Owner**: Definiert die übergreifenden Ziele des Projekts und kümmert sich um den ROI
  - **Scrum master**: Leitet den gesamten Prozess und sorgt für die Einhaltung der Spielregeln (üblicher weiße kein Entwickler)
- Beachtenswert: Make SMART requirements – INVEST in user stories – make sure a task is TECH

Kapitel 7 – Die Mircosoft-Methode

Denkweise des Industriezweigs "**shrink-wrap software**":

1. Shrink wrap software has its own laws (keine Haftung für nicht funktionierende Software)
2. PC software is a way of life (viele neue Versionen veröffentlichen)
3. Get your team into ship mode (es wird programmiert um zu Verkaufen)
4. Time to market is more important than product quality

Allgemein:

- Kleine feature groups von 3-8 Entwicklern. (+ 3-8 Tester)
- Glauben verbreiten, dass sie nur die Besten einstellen und alle angestellten die Besten sind.
- Die "**sync and stabilise**"-Methode: daily build + Regressionstest für den ein Mitarbeiter (build master) verantwortlich ist. Mindestens alle 2 Tage wird der Quellcode eingecheckt, dadurch gibt es Zwischenversionen.
- Jeden Morgen wird die erzeugte Zwischenversion von einem Usability Labor übernommen, die die Anwender simulieren.

7.1 Vergleich mit XP

- Weitestgehend gibt es nur im Detail Unterschiede.
- **Einfacher Entwurf**: ist bei Mircosoft nicht vorhanden
- **Pair Programming**: Nicht vorhanden, da gewinnorientiertes Unternehmen.
- **Collective code ownership**: Es gibt für jede Komponente einen Hauptverantwortlichen bzw. eine verantwortliches Team.

Kapitel 8 – Leistungsverbesserung von Software

Beste Strategie zur Leistungssteigerung:

1. Benutze die einfachsten, saubersten Algorithmen und Datenstrukturen, die für die Aufgabe geeignet sind.
2. Miss die Leistung um zu sehen, ob Änderungen nötig sind.
3. Schalte Compiler-Optionen ein, um den schnellstmöglichen Code zu erzeugen.
4. Stelle fest, welche Änderungen am Programm die größte Wirkung haben werden.
5. Führe die Änderungen jeweils einzeln durch und miss jeweils erneut.
6. Hebe alle Versionen auf, um Revisionen dagegen zu testen.

Grobe Unterscheidung der Profiler:

- Line count profiler: Zählen, wie oft eine Zeile ausgeführt wurde
- Run time profiler: Zählen, wie oft eine Prozedur aufgerufen wurde (inkl. Zeit)

8.1 Fallstrike

- Verwechslung von Leistung mit Durchsatz. (Parallelisierung beachten)
- Durchschnittliche Antwortzeit sagt nichts aus, falls sich die Antwortzeiten stark streuen (**Perzentil-Spezifikation** z.B. 90% der Anfragen werden in einer Sek. oder schneller erledigt)
- **Amdahls Gesetz**: Es bietet sich unter anderem aufgrund der Kosten einer Verbesserungsmaßnahme an eine billigere Maßnahme mit weniger Wirkung gegenüber einer teuren Maßnahme mit größerer Wirkung zu bevorzugen.

Kapitel 9 – Softwarewerkzeuge

9.1 awk, grep, sed

- **sed**: ist der Stream Editor. Er liest eine Eingabedatei Zeichen für Zeichen und schreibt dabei gleichzeitig eine Ausgabedatei (keine Größenbeschränkung von Dateien oder Zeilen)
- **grep**: Eine Abkürzung von "global/regular expression/print". Sucht in einer Textdatei nach Zeilen, die ein bestimmtes Pattern beinhalten, das durch einen regulären Ausdruck definiert ist.
- **awk**: kann komplexe Manipulationen auf Textdateien vornehmen. Zerlegt Zeilen in Felder die durch (normalerweise Leerzeichen) getrennt sind. Führt dann Aktionen auf Zeilen aus in denen ein gewisses Pattern vorhanden ist.

9.2 Das Werkzeug "make"

- Liest aus einem sogenannten makefile Beschreibungen der Abhängigkeiten zwischen Dateien und führt daraufhin alle jeweils notwendigen Aktionen durch.

9.3 Das Werkzeug "Ant"

- Eine Art neuerer Ansatz für make der Betriebssystem unabhängig ist und geradezu auf Java zugeschnitten ist. Beschreibungen stehen in einer build.xml

9.4 Konfiguration von Software

- Plattform ist eine Kombination von Maschine und Betriebssystem. Der allgemeinere Begriff Substrat bezieht auch noch die GUI mit ein.
- Je nach Substrat gibt es womöglich andere Optionen für Compiler, Linker und andere Pfade zu Programmbibliotheken und diese beeinflussen das makefile.
- **Präprozessor**: Mechanismen zur bedingten Kompilation, wird dazu verwendet verschiedene leistungsfähige Versionen des gleichen Programms (z.B. Evaluationsversion, Vollversion)
- Software für viele Plattformen verfügt deshalb über rech elaborierte configure-Skripte. (GNU)
- **Configure-Skripte** leisten im wesentlichen das folgende:
    - Prüfen der vorliegenden Plattform, d.h. Hardware, Betriebssystem, Compiler
    - Auffinden der Suchpfade für Programmquellen und Bibliotheken
    - Auffinden weiterer wichtiger Details des Substrats
    - Setzen aller relevanten Präprozessor-Variablen
    - Erzeugung von makefiles aus vorgegebenen Rahmen

9.5 Versionshaltung mit SVN

- **SCCS** (Source Code Control System): Urvater aller Versionsverwaltungen, verwaltet nur einzelne Dateien!
- **CVS** (Concurrent Verions System): Probleme im Umgang mit Binärdaten

Zwei unterschiedliche Strategien:

- **Zentrales Repositorium** (SVN, CVS): Ein einziges zentrales Repositorium und Entwickler ziehen sich Arbeitskopien. Verschiedene branches sind für alle sichtbar.
- **Verteilte Repositorien** (Mercurial, GIT): Jeder Entwickler hat ein eigenes komplettes Repositorium. Eine Kopie erhält er von einer anerkannten Quelle (clone). Innerhalb seines eigenen Repositorys kann er bliebig committen und nach fertigstellung pushen. Er sollte allerdings erst mit pull prüfen

9.6 Werkzeuge zur Dokumentation

9.6.1 Nroff

- wird vor allem für UNIX-Programme (Manual-Pages) verwendet
- im Prinzip eine Textdatei die mit Kommandos (.SH (section heading), .PP, .NH, .br, .bp) durchsetzt ist und von dem Tool nroff interpretiert wird

9.6.2 Texinfo

- Zum Editor Emacs gehört das Werkzeug
- Doppelte Zielrichtung: Einerseits Hypertexte andererseits kann man auch über Umweg über TEX ein gedrucktes Handbuch erstellen

9.6.3 Javadoc

- Single source-Prinzip (Vorteil der Konsistenz der Dokumentation)

Kapitel 10 – Free/Libre/Open Source Software

- die frühen Computer waren sämlich in den Händen von Wissenschaftlern
- Wissenschaftler publizieren ihre Erkenntnisse, die dann von anderen Wissenschaftlern frei verwendet, verbessert und weiterentwickelt werden können
- Durch einen Prozess gegen IBM hat der amerikanische Gerichtshof sozusagen die Softwareindustrie gegründet indem IBM gezwungen wurde ihre Software zu verkaufen
- Richard Stallman gründete das GNU-Projekt ("GNU's Not Unix")
    - Die Freiheit, das Programm zu jedem beliebigen Zweck laufen zu lassen
    - Die Freiheit, zu studieren, wie das Programm funktioniert, und es an die persönlichen Bedürfnisse anzupassen.
    - Die Freiheit, Kopien weiter zu verteilen, um dem Nachbarn zu helfen
- Die Freiheit, das Programm zu verbessern, und die Verbesserungen zu publizieren
- Von Free Software sollte nur dann gesprochen werden, wenn sie unter der GPL vertrieben wird.
- BSD-lizenz stellt lediglich klar, dass der Hersteller nicht für irgendwelche Schäden haftet und die Copyright-Notiz unverändert erhalten bleibt
- Public Domain sagt nicht aus, außer dass der Quellcode der Öffentlichkeit zur Verfügung stellt, d.h. auf seine Besitzrechte verzichtet. (Jeder kann damit tun was er will)