Zusammenfassung Software-Technik

Im Rahmen der Vorlesung Software Technik WS 2011/2012
Bei Prof. Dr. M. Leucker

Jasper Schwinghammer, Raphael Allner

Vorwort

Version 0.7 25.02.2012
Autoren: Jasper Schwinghammer, Raphael Allner, Studenten der Informatik Universität zu Lübeck
Textsatz mit LaTeX
Alle Bilder sind geistiges Eigentum Ihrer Eigner.

Inhaltsverzeichnis

1 Grundlagen

1.1 Etymologie

Software Engineering = Softwareerstellung nach Ingenieurmethoden.
Nach Boehm von 1979 ist Software Engineering die praktische Anwendung wissenschaftlicher Erkentnisse auf den Entwurf und die Konstruktion von Computerprogrammen.

Software Engineering ist also allabout wissenschaftliche Prinzipien wie Software entwickelt werden sollte und wie dies in der Praxis umgesetzt wird. Hierbei wird auch auf zeitgerechte Fertigungsprozesse, sowie die Wartung und der Lebenszyklus berücksichtigt. Alles was in Softwaretechnik gelehrt wird setzt sich mit diesen Thematiken auseinander.

„Software Engineering ist die praktische Anwendung wissenschaftlicher Erkenntnisse zur wirtschaftlichen herstellung von qualitativ hochwertiger Software und für deren wirtschaftlichen Einsatz.“

1.2 Themenbereiche der Software Technik

1. Anforderungen
2. Spezifikation
3. Beherrschung der Komplexität
4. Schnittstellen
5. Wiederverwendung
6. Projektorganisation, Arbeitsteilung
7. Dokumentation
8. Entwurf, Test,
9. Einführung, Betrieb, Wartung
10. Änderungen, Erweiterungen, Anpassungen

1.3 Trivia

In dieser Sektion finden sich anfangs erstmal nur triviale Dinge die am Anfang der Vorlesungsreihe von Leucker erwähnt wurden.

1.3.1 Kosten

Die Softwarekosten sind in den vergangenen 27 Jahren um schätzungsweise 1000% angestiegen, gleichzeitig sind die Kosten für Hardware gesunken.

1.3.2 Marktfaktoren

Bei der Ertragsstärke wird zwischen Standartsoftware die große Verbreitung und einfache Anwender hat (Excel) und teurer Spezialsoftware mit anspruchsvollen Anwendern (z.B. Maya) unterschieden.

1.3.3 Qualität

Qualitätsmerkmale und Qualitätssicherung - to be done

1.3.4 Produktivität

Ausbildungsstand, Anwendungskompetenz Problemkomplexität, Unterstützungswerkzeuge - to be done (Herczeg?)

1.4 Phasen der Softwareerstellung

Die Softwareerstellung gliedert sich nach 8 Phasen.

1.4.1 Anforderungsphase, Anforderungserhebung

In der Anforderungsphase werden die Anforderungen an das System mit den Kundenwünschen und dem Anwendungsgebiet abgestimmt. Hierbei sollte man differenzieren ob man eine Spezialsoftware für einzelne Kunden entwickelt, oder ob man eine Standardsoftware für einen breiten Markt erstellt. Muss man also den Ansprüchen eines einzelnen Kunden entsprechen und primär auf diesen eingehen oder muss die Software für sehr viele Zielgruppen von Nutzern ansprechend sein. Letzteres erfordert eine weitaus komplexere Marktanalyse und außeinandersetzen mit dem Ausbildungsstand sowie der Einstiegsfreundlichkeit der Software.
Man spricht in diesem Zusammenhang auch von der Systemanalyse.
Wikipedia:

> Im Zusammenhang mit der Projektentwicklung ist hier die Systemanalyse zur Projektvorbereitung gemeint. Gegenstand ist die inhaltliche Erfassung der Anforderungen durch Befragung künftiger Anwender sowie die systematische

Untersuchung weiterer sachlicher und technischer Anforderungen und Randbedingungen (Schnittstellen zu Drittsystemen, gesetzliche Anforderungen u.dgl.). Ergebnis ist meist ein Fachkonzept.

1.4.2 Spezifikationsphase

Nach der eingehenden Analyse wird nun in der Spezifikationsphase die Funktionen und Leistungen des Softwareprodukts präzise festgelegt. Normalerweise werden die Spezifikationen in einem Spezifikationsdokument festgehalten dem sog. Kontrakt.

1.4.3 Planungsphase

Die Planungsphase legt sozusagen die Roadmap des Projektes fest. Hier werden Raum-, Ort-, Terminplanung des Projektmanagements, Ablaufpläne, Finanzierungspläne und die Meilenstein-Feinplanung festgelegt. (Kosten-Nutzen-Schätzung)

1.4.4 Entwurfsphase

Die Entwurfsphase spezifiziert das Projekt weiter. Sie besteht im weiteren Sinne daraus, zunächst grob Diagramme anzufertigen und dann bis ins Detail die Systemarchitektur und Abläufe innerhalb des Systems festzulegen (UML, Petrinetze). Hardwareentwürfe werden ebenfalls hier gemacht, sofern diese benötigt werden. Man zerlegt in dieser Phase die Systemarchitektur in Module und diese werden anschließend im Detail geplant bzw. die Implementierung dieser.

1.4.5 Implementierungsphase

In der Implementierungsphase werden die Programmkomponenten (Unterprogramme/-Module) implementiert und getestet. Der Test ist hier in zweierlei Hinsichten zu sehen. Einerseits hinsichtlicht der korrekten Umsetzung (keine Fehler) andererseits inhaltlich im Bezug auf den Kontrakt ob die Implementierten Funktionen sich für ihren vorgesehenen Zweck eignen.

1.4.6 Integrationsphase

In der Integrationsphase wird das Gesamtsystem zusammengefügt und getestet, sowie die Dokumentation fertig gestellt. Sie endet mit der Auslieferung an den Kunden.

1.4.7 Wartungsphase

Die Wartungsphase ist der teuerste Teil eines Softwareprojekts, daher ist bei der Implementierung auf gute Wartbarkeit, Erweiterbarkeit und Dokumentation zu achten. Die Wartungsphase besteht aus dem Betrieb beim Kunden, der Fehlerkorrektur (Bugs beheben), der Erweiterung der Software sowie die Fehlerbehebung.

1.4.8 Rückzugsphase

Die Rückzugsphase ist die letzte Phase im Softwareentwicklungszyklus, hier wird die Nutzung der Software eingestellt und die Daten gesichert oder in die neue Software portiert.

2 Softwarequalität

2.1 Qualitätskriterien

2.1.1 Standpunkt des Benutzers: Akzeptanz

Effektivität

Die Effektivität setzt sich aus Komfort (funktioneller Umfang) Korrektheit (leistet Gewünschtes) und Konsistenz (keine Widersprüche) zusammen.
Nach ISO 9241-11 drückt Effektivität

> Die Genauigkeit und Vollständigkeit mit der Benutzer ein bestimmtes Ziel erreichen.

aus. [Her09]

Effizienz

Effizienz betrifft Hauptsächlich die Ressourcen-Nutzung von Software die für den Nutzer merkbar ist. Sprich hauptsächlich Speicherbedarf, Laufzeit und Antwortzeit. Auch die Energieffizienz spielt eine Rolle heutzutage.

> Der im Verhältnis zur Genauigkeit und Vollständigkeit eingesetzte Aufwand mit dem Benutzer ein bestimmtes Ziel erreichen

[Her09]

Zuverlässigkeit

Zuverlässigkeit setzt sich aus relativ intuitiven zusammen.

1. Integrität und damit der Erhalt von Programm und Daten sowie die Abweisung ungültiger Eingaben
2. Redundanz Das System sollte über zumindest über etwas mehr Ressourcen verfügen als es im Normalfall braucht, außerdem sollten Datensätze doppelt geführt werden.
3. Sicherheit im Sinne von Störungsschutz und vermeidung von Datenverlust
4. Wiederherstellbarkeit Zeitdauer bis System nach Fehler oder Störung wieder funktioniert
5. Verfügbarkeit Wahrscheinlichekit für funktionsfähigen Zustand, Verhältnis von Fehlzeit zu Betriebszeit

2.1.2 Standpunkt des Produzenten: Ausbaufähigkeit

Wartungsfreundlichkeit

Die Wartbarkeit eines Systems hängt elementar von der Implementierung ab. Die Implementierung muss Kontrollierbar, Reparierbar, Transparent, Modular und Normgerecht sein.
Die Kontrollierbarkeit wird von der Protokollierung bestimmt, es ist deutlich einfacher ein System zu warten das aktuelle und vergangene Systemzuständige speichert. Beispiel Fressautomat Bahnhof, durch eine Glasscheibe kann man sehen welche Produkte noch da sind → Extrem Wartunsfreundlich ;).
Reparierbarkeit eines Systems ist elementar davon abhängig wie gut sich Fehler lokalisieren lassen. Das Beheben ist dann je nach Lokalisierung des Fehlers aufwendig oder nicht. Durch die Modularität der Software muss schlimmsten Falls ein Modul ausgetauscht werden.
Die Transparenz einer Software bezieht sich auf die Verständlichkeit, Genauigkeit und Vollständigkeit eines Codes. Die Dokumentation spielt hier eine elementare Rolle.
Modularität bezieht sich auf die Strukturierung des Softwaresystems in Bausteine, die Wartung des Systems ist deutlich leichter wenn der Code modular aufgebaut ist.
Normgerechtigkeit hängt indirekt mit der Transparenz zusammen, wenn sich an allgemeine oder firmenspezifische Normen gehalten wird ist die Lesbarkeit und die Wartbarkeit eines Programms deutlich verbessert und die zusammenarbeit mit Software Dritter ist möglich.

Anpassungsfähigkeit

Grad der Flexibilität, um ein System an geänderte oder erweiterte Anforderungen anzupassen

Maschinenunabhängigkeit

Die Maschinenunabhängigkeit setzt sich aus Portabilität (Übertragbarkeit von einem System uns andere) und Kompatibilität (Verträglichkeit mit Hard- und Softwareeinheiten, Austauschbarkeit und Herstellerunabhängigkeit) zusammen.

2.1.3 Konflikte

Effizienz und Maschinenunabhängigkeit sind im weitesten Sinne widersprüche, da Effiziente Programmierungen meistens Hardwarenah sind, Maschinenunabhängigkeit aber eben nicht-hardwarenahe Programmierung voraussetzt.
Softwarekosten wollen gering gehalten werden, mit der Softwarequalität steigen aber meist auch die Softwarekosten.

Der Softwareentwickler muss hier ein passenden Kompromiss eingehen.

2.2 Qualitätssicherung

Die Softwarequalität soll durch Maßnahmen **während** der Softwareerstellung gesichert werden.

2.2.1 konstruktive Maßnahmen

Um zu gewährleisten, dass die Softwarequalität entsprechend hoch ist, sollten während der Entwicklung alle festgelegten Kriterien eingehalten werden (Pflichtenheft, Nord, siehe Qualitätskriterien). Ebenfalls sollte drauf geachtet werden, dass die Umgebung in der das Projekt entsteht entsprechend gut ist (nicht in BlueJ programmieren). Des weiteren sollten die verwendeten Drittbibliotheken o.Ä. qualitativ hochwertig sein.

2.2.2 Analytische Maßnamen

1. statische Programmanalyse
2. dynamische Programmanalyse
3. systematisches Testen

2.2.3 Organisatorische Maßnahmen

Vorgehensmodelle, weiterbildung, Institutionalisierung der Qualitätssicherung (eigene Mitarbeiter dafür).

3 Vorgehensmodelle

Wir verzichten auf das Pro und Contra-Schema der Vorlesung weil die sich regelmäßig widersprechen.

3.1 Lebenszyklusmodell

In dem Lebenszyklusmodell sind alle phasen klar gegliedert und voneinander getrennt. Erst wenn eine Phase komplett abgeschlossen ist, wird in die nächste Phase des Projektes gegangen.
Erst wenn der Lebenszyklus durchlaufen ist, können neue Features eingebaut werden.

3.1.1 Eigenschaften

- Die Entwicklungschritte sind klar getrennt voneinander
- sequenztiell, keine Möglichkeit zur Rückkopplung
- Wechselwirkungen zwischen Phasen können hier nicht berücksichtigt werden, Bewegung im Zyklus geht nur in eine Richtung
- Sehr strukturiert und Projektgrößen unabhängig

– Insert Picture

3.2 Wasserfallmodell

Das Wasserfallmodell entspricht dem Lebenszyklusmodell, bis auf, dass Rückkopplungen zwischen aufeinander folgenden Phasen möglich sind, und Validierungen und Tests in jeder Entwicklungsphase gemacht werden.

3.3 Prototyp-orientierte Modelle

Das Prototyp-orientierte Modell ist iterativ. Die Phaseneinteilung bleibt im wesentlichen bestehen, es darf aber Überlappungen geben. Hierbei wird nicht nach jeder Phase getestet sondern eher in Blöcken gedacht (z.b. Entwurf und Spezifikation) man spricht hierbei auch von Verschmelzung zusammengehöriger Phasen. Vom jeweiligen Prototyping kann dann in Phasen zurückgesprungen werden, je nachdem ob der Prototyp zufriedenstellend ist.

Vergl Lebenszyklusmodell: Lebenszyklus: so spät wie möglich Prototyping: so früh wie möglich implementieren.

3.4 Spiralmodell

Das Spiralmodell ist eine Variante des Lebenszyklusmodell.
Jeder Zyklus besteht aus: Festlegung der Ziele, Erarbeitung von Lösungsvarianten, Entwicklung einer Lösung, Planung der nächsten Phase. Jede Projektphase entspricht hierbei einem Zyklus.
Durch eine Risikoanalyse in jeder Phase wird das Softwareentwicklungsrisiko vermindert.
Dieses Entwicklungsmodell wird primär auf großprojekte angewendet, für kleine rechnet sich der Aufwand nicht.

3.5 Inkrementelle Methoden

Iteration über die Entwicklungsphasen: Analyse - Spezifikation - Entwurf - Implementierung - Test

3.6 Extreme Programming

Ist eine Variante der inkrementellen Methoden hierbei wird in Zweiergruppen mit öffentlichem Code programmiert und sehr eng mit Benutzern zusammengearbeitet.
Eignet sich nur für kleinere Projekte, benötigt teamfähige Entwickler. Es wird allerdings wenig dokumentiert.

4 Anforderungsfestlegung

4.1 Definition

Vom Englischen *Requirements Engineering* (zu deutsch Anforderungsfestlegung) spricht man von der Auseinandersetzung mit der systematischen, ingenieurmäßigen Entwicklung einer Anforderungsdefinition, welche die Leistung eines Systems vollständig und eindeutig beschreibt.

4.2 Anforderungsdokument

Durch eine Problemanalyse, eine Anforderungsdefinition und eine Anforderungsanalyse hat die Anforderungsfestlegung als Ergebnis ein Anforderungsdokument(Auch bekannt als: Pflichtenheft, Produktdefinition und Requierments-Spezifikation)

4.2.1 Funktion eines Anforderungsdokuments

In dem Anforderungsdokument wird der Kontrakt(dummes Wort für Vertrag) zwischen Auftraggeber und Auftraggeber niedergelegt. Hinzu kommen alle von den Benutzern geäußerten Wünsche hinsichtlich des Produkts. Die Spezifikation und der Grobentwurf liegen ebenfalls bei.

4.3 Anforderungsgruppen

- **Funktionale Anforderung („Was“):** Welche Funktionen soll das System haben und wie ist das Ein- und Ausgabeverhalten, also wie soll es für den Benutzer und wie für andere Gruppen aussehen.
- **Qualitätsattribute gewünschter Funktionen („Wie “):** Wie ist das Ausführungsverhalten festzulegen, also wie lange ist die Antwortzeit, wie genau ist die Ausgabe und wie groß ist der Speicherbedarf. Wie Zuverlässig und Ausfallsicher muss das System sein?
- **Anforderungen an das Gesamtsystem:** Allgemeine Beschreibung wofür das System da ist.
- **Vorgaben für die Durchführung:** Wie wird vorgegangen welche Modelle werden benutzt bla bla bla.....

- **Vorgaben an Installation und Betrieb:** Zitat jasper: Alleine für die Überschrift würde ich das weg lassen.
- **Prinzipielle Vorgehensweise:** Ermittlung - Beschreibung - Analyse - Abnahme

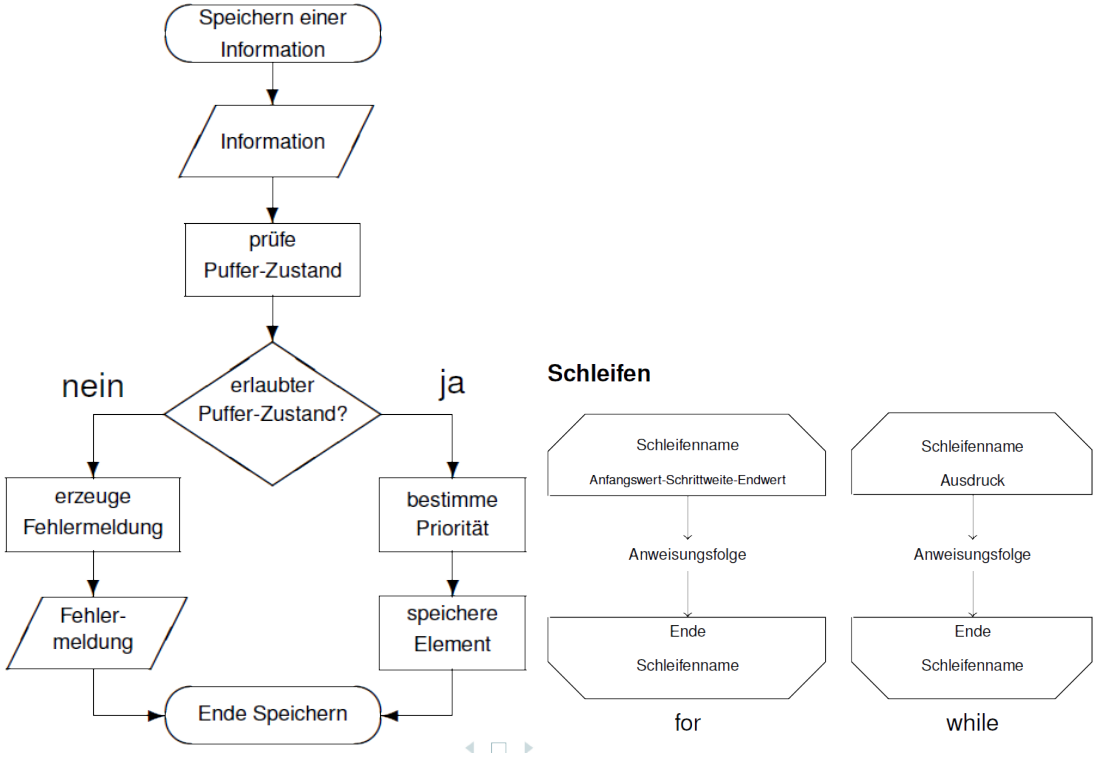
5 Algorithmische Sicht

5.1 Programmablaufpläne

Programmablaufpläne (Flussdiagramme, Ablaufdiagramme) sind Graphen die Anhand von Knoten und gerichteter Kanten Verarbeitungsschritte beschreiben. Diese Schritte nennt man auch den Kontrollfluss.

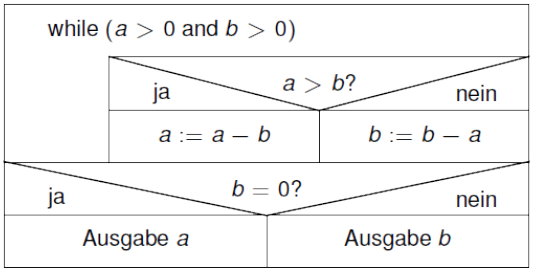
Die Nachteile eines Programmablaufplans: Bei einem Ablaufplan fehlen viele Informationen da Ausschließlich der Ablauf beschrieben wird und nicht differenziert auf die Datenstrukturen eingegangen wird. Für große komplexe Systeme eignet sich diese Methode also nicht, da nur wenig Möglichkeiten zu Verfeinerung und Abstraktion zur Verfügung stehen.

5.1.1 Beispiel



Struktogramme

Struktogramme ermöglichen eine gute grafische Darstellung von linearen Kontrollstrukturen während Sprünge im Plan nicht vorgesehen sind. Sie sind aufwendig zu zeichnen und zu ändern was aber durch die Digitalisierung entfällt. Sie sind besser zu lesen als Programmablaufpläne.



6 Funktionale und datenorientierte Sicht

6.1 Funktionsbäume

Funktionsbäume dienen zur systematischen Gliederung einer Vielzahl von Funktionen und geben erste Hinweise für Implementierung und Dialoggestaltung. Dabei beschreiben die Blätter oder Unterbäume die Teilfunktionen einer Funktion X.

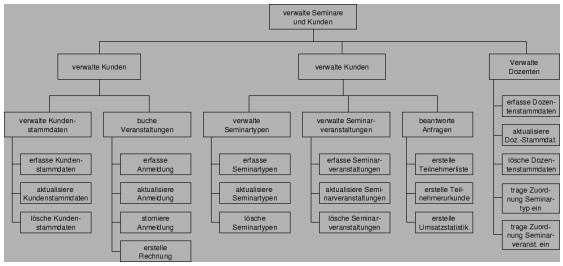


Abbildung 6.1: Funktionsbaum Beispiel

6.2 Datenflussdiagramme

Datenflussdiagramme beschreiben den Informations-/Datenfluss zwischen den Komponenten eines informationsverarbeitenen Systems. Bei großen Systemen sind diese Graphen jedoch sehr unübersichtlich und eine differenzierte Festlegung bestimmter Funktionen fällt genauso wie eine differenzierte Darstellung der Datenstrukturen aus.

6.2.1 Syntaktische Regeln

- Es existiert mindestens eine Schnittstelle.
- Jede Schnittstelle ist nur einmal vorhanden. Wird ein Diagramm zu unübersichtlich, dann kann eine Schnittstelle auch mehrfach gezeichnet werden.
- Jeder Datenfluss hat einen Namen.
- Zwischen Schnittstellen gibt es keinen Datenfluss.
- Zwischen Speichern gibt es keine direkten Datenflüsse.
- Zwischen Schnittstellen und Speichern dürfen keine direkten
- Datenflüsse gezeichnet werden.

6.2.2 Arten von Datenflussdiagrammen

Es gibt verschiedene Notationen zur Darstellung von Datenflussdiagrammen. Die aktuellere Notation ist jene von Tom DeMarco. Früher wurden die Symbole aus DIN 66001 verwendet. Diese Notation ist heute aber eher unüblich.

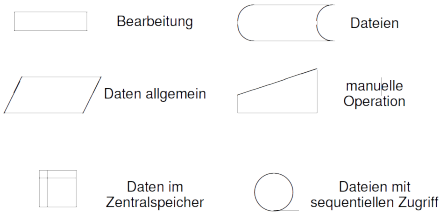


Abbildung 6.2: Symbole nach DIN 66001

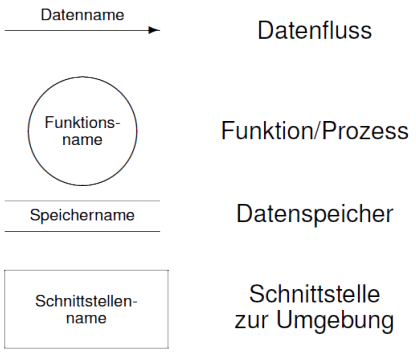


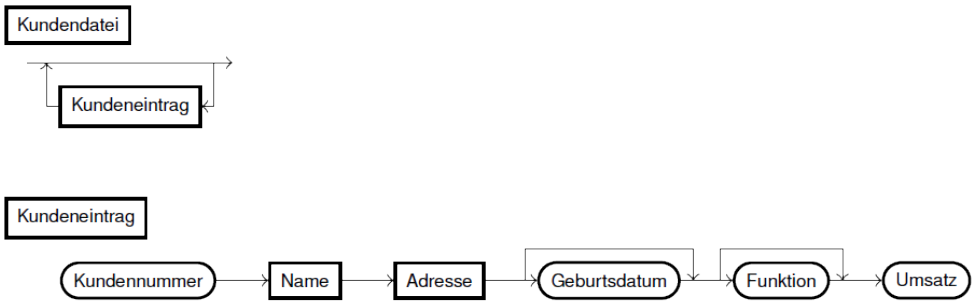
Abbildung 6.3: Symbole nach DeMarco 1978

6.3 Datenkataloge

Datenkataloge(auch Datenlexikon) sind Verzeichnisse, die die vorkommenden Datenelement eines Systems anhand einer Notation, welche erweiterten Backus-Naur-Form genannt wird oder durch Syntaxdiagramme beschreiben. Beides erfüllt den gleichen Zweck sieht aber anders aus.

=	ist definiert durch
+	Sequenzbildung (“und”)
[|]	Auswahl (“ entweder ... oder”)
{ }	Wiederholung
M{ }*N*	Wiederholung von *M* bis *N*
( )	Option
...	Kommentar

```
Kundendatei    = {Kundeneintrag}
Kundeneintrag = Kunden-Nr. + Name + Adresse
                + (Geburtsdatum) + (Funktion) + Umsatz
Name           = Anrede + (Titel) + Vorname + Nachname
Adresse        = [Strasse + Haus-Nr. | Postfachnummer]
                + (Laenderkennzeichen) + PLZ + Ort
                + (Telefon) + (Fax)
```



6.4 Jackson-Diagramme

Jackson-Diagramme Jackson-Diagramme sind eine einheitliche, graphische Baumdarstellung für Daten- und Kontrollstrukturen, die aus Sequenzen, Auswahlen und Wiederholungen besteht. Bei diesen Bäumen werden Sequenzen so dargestellt, dass Unterbäume/Blätter zeitlich von links nach rechts abgearbeitet werden. Bei einer Auswahl, also if Abfrage oder switch bekommen die Blätter ein Kreis oben rechts. Bei Wiederholungen erholten die Blätter Sternchen oben rechts.

Jackson Diagramme sind den Datenkatalogen ähnlich können aber die Anzahl einer Wiederholung nicht einschränken. Sie sind gut lesbar, haben aber eine platz raubende Darstellung.

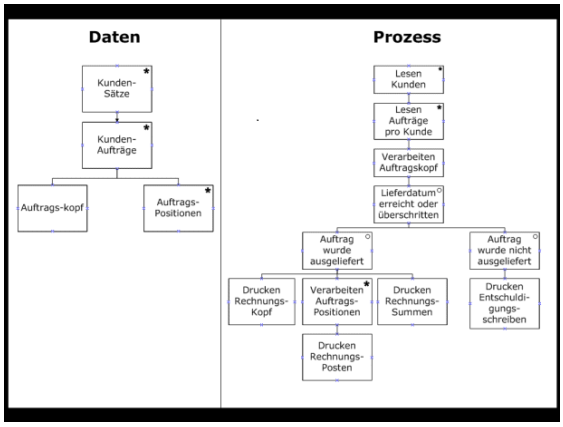


Abbildung 6.4: Jackson Diagramm Beispiel

6.5 Entitäten und Beziehungen

Das **Entity-Relationship-Modell** wurde ursprünglich zur Datenmodellierung beim Datenbankentwurf verwendet (Chen 1976).

Dabei werden Systeme durch Entitäten(Objekte/wohl unterscheidbare Dinge) und deren Beziehungen zueinander modelliert. Weiter werden alle Attribute(Eigenschaften) der Objekte und der Beziehungen beschrieben. Des Weiteren werden die einzelnen Entitäten und die Beziehungen zu Entitätsmengen und Beziehungsmengen zusammengefasst.

Der Vorteil einer solchen Modellierung liegt darin das sie schnell erlernbar ist und flexibel eingesetzt werden kann. Es können alle Arten von Systemen modelliert werden, was der großen Allgemeinheit zuzuschreiben ist. Man kann jedoch keine Abläufe beschreiben wodurch die methodische Unterstürzung, also der Zweck, bei der Erstellung eines komplexen Systems fraglich bleibt. Diese Art der Modellierung eignet sich lediglich für Datenbanken und Informationssysteme.

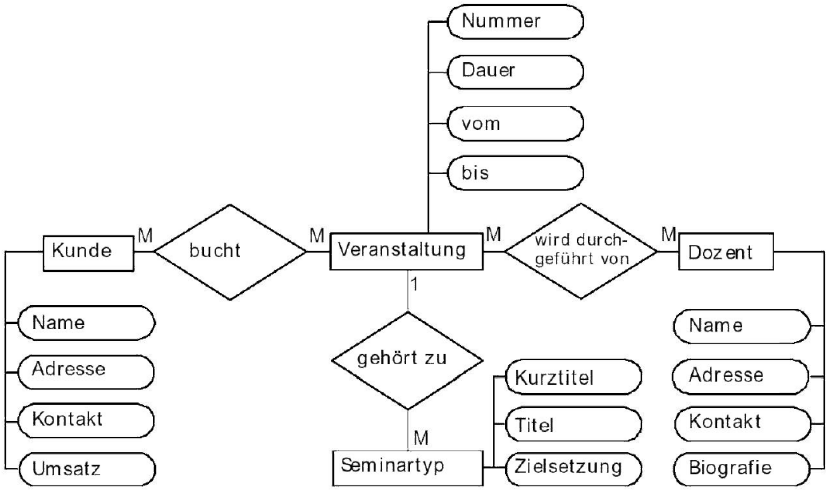


Abbildung 6.5: Entität und Beziehungen Beispiel

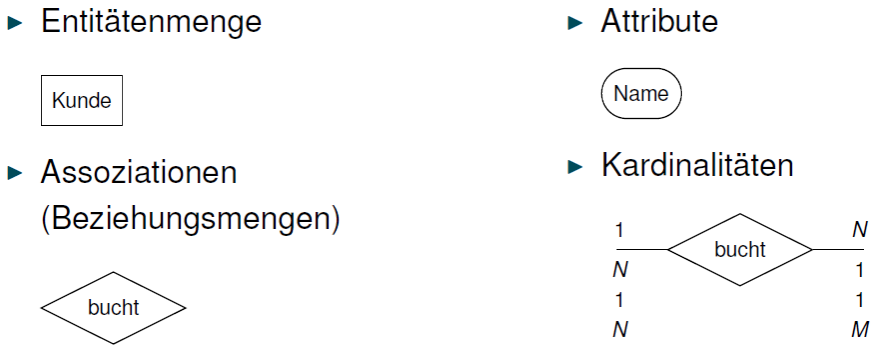


Abbildung 6.6: Entität und Beziehungen Notation

7 Ablaufsicht

7.1 Petri-Netze

Petri-Netze dienen der ablauforientierten Beschreibung von nebenläufigen und verteilten Systemen. (Ereignisse sind nebenläufig, wenn keines eine Ursache des anderen ist.) Sie kommen zum Einsatz wenn es um Prozessmodellierung oder Ablaufsbeschreibungen geht. Dabei können endliche und unendliche Abläufe beschrieben werden, sowie Systeme auf einen möglichen Deadlock oder bestimmte erreichbare Zustände überprüft werden können.

Raphael:*Das ist meine eigene Beschreibung wie PetriNetze aufgebaut sind und wie sie funktionieren, wem das zu ungenau und zu informell ist kann gerne im offiziellen Skript nachlesen.*

Ein Petri Netz(Netzgraph) besteht aus einer Menge **S(manchmal auch P) von Stellen**, Marken, gerichtete Kanten, deren **Gewichtung W**, der **Flussrelation F**, und einer **Menge T von Transitionen**. Die Stellen sind im mit einer bestimmten Anzahl von Marken gefüllt und man unterteilt verschiedene Mögliche Konstellationen in Zustände welche man auch die **Markierung** M_n(Belegung der Stellen oder Konfiguration) nennt.

Die Stellen deren Kanten von der Stelle auf die Transition zeigen nennt man den**Vorbereich** während man die Stellen auf die die Transition zeigt den **Nachbereich** nennt. Je nach Anzahl der Marken in einer Stelle im Vorbereich wird die Transition aktiviert wodurch in den Stellen im Nachbereich neue Marken generiert werden. Die Benötigte Anzahl an Marken und die generierte Anzahl an Marken wird durch die Gewichtung der Kanten definiert. Man spricht davon, dass eine Transition zwischen 2 Markierungen schaltet und wenn eine bestimmte Markierung nur über mehrere solche Schaltvorgänge erreichbar ist, nennt man das auch eine **Schaltfolge**.

Zum Vorgang: Nehmen wir an, es gibt eine Transition, die erst aktiviert wird, wenn sie Zugriff auf 2 Marken an 2 verschiedenen Kanten hat. Diese Anzahl wird an den Kanten die auf die Transition zeigen notiert. Wenn also eine oder mehrere Stellen die für die Schaltung benötigte Anzahl an Marken beinhalten, schaltet die Transition und generiert neue Marken in den Stellen des Nachbereichs, wodurch in die nächste Markierung gewechselt wird. Wie viele Marken in den Stellen generiert werden, auf die die Transition mit einer Kante zeigt, hängt wiederum von der Gewichtung dieser Kanten ab.

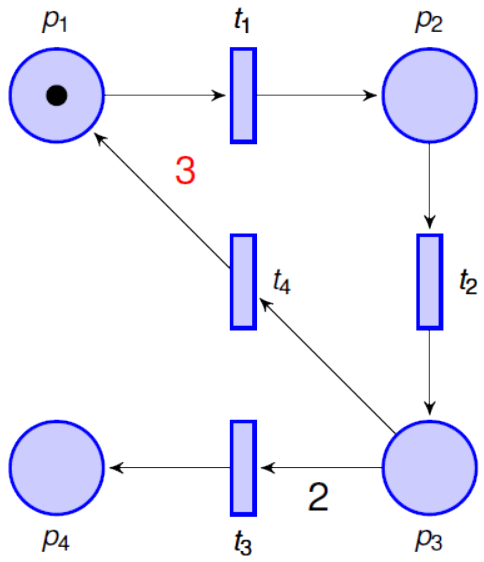
7.1.1 Erreichbarkeit

Es geht um die Erreichbarkeit einer bestimmten Markierung. Heißt ist eine bestimmte Markierung über eine bestimmte Schaltfolge erreichbar. Dazu wird ein Graph gezeichnet bei dem alle möglichen Markierungen als Knoten dargestellt werden und durch Kanten welche mit den jeweiligen Transitionen gekennzeichnet sind miteinander in Beziehung gesetzt. Diesen Graph nennt man dann **Ereichbarkeitsgraph**.
Z.B. kann ich eine Konfiguration A über die Transition T1 von der Konfiguration B aus erreichen. Es gibt verschiedene Möglichkeiten dies Graphisch darzustellen. Zum einen kann man jede mögliche Markierung mit Buchstaben oder Zahlen codieren oder man schreibt die Konfiguration direkt als Vektor in die Knoten des Erreichbarkeitsgraphen.

Meistens ist ein solcher Erreichbarkeitsgraph nicht endlich und deshalb irrelevant. Interessant wird es erst wenn es zu einer Verklemmung kommen kann, also ein Zustand ab dem keine Transition mehr möglich ist.

7.2 Anwendungen

Wenn an einer Kante nichts angegeben ist, ist ihre Gewichtung/Kapazität 1. Wenn nicht anders angegeben ist die Kapazität einer Stelle ∞.



7.3 Beispiel Aufgabe zu Petri-Netzen

Betrachten Sie das Petri-Netz $N = (P, T, F, W, M_0)$ mit

- $P = \{p_1, p_2, p_3\}, T = \{t_1, t_2, t_3, t_4\}$
- $F = \{(p_1, t_1), (p_1, t_2), (t_2, p_1), (p_2, t_2), (t_2, p_3), (p_2, t_3), (t_3, p_3), (p_3, t_4), (t_4, p_2)\}$
- $W = \{(p_2, t_3) \rightarrow\}\{f \rightarrow 1 | f \in F\text{"} \ \{(p_2, t_3)\}\} \cup \{f \rightarrow 0 | f \notin F\}$
- $M_0 = \{p_1 \rightarrow 1, p_2 \rightarrow 2, p_3 \rightarrow 0\}$

Zeichnen Sie eine graphische Repräsentation des Netzes N und seines Erreichbarkeitsgraphen. Kann N sich ausgehend von der Startkonfiguration verklemmen? Falls ja, geben Sie alle erreichbaren Verklemmungen und jeweils eine zugehörige Schaltsequenz von Transitionen an.

8 Zustandsorientierte Sicht

8.1 Zustandsorientierte Modellierung

Bei der Zustandsorientierten Modellierung kann eigentlich als bestes Beispiel die uns allen bekannten Moore Automaten genannte werden. Wie diese funktionieren und was sie modellieren sollte jedem mittlerweile klar sein.

8.2 Zustandsübergangsmaschine

Behinderter Scheißdreck ...

8.2.1 Definition

Eine **Zustandsübergangsmaschine** (state transition machine)

$$M = (State, In, Out, next, out, q_0)$$

besteht aus einer **Zustandsmenge State**, einer **Eingabemenge In**, einer **Ausgabemenge Out**, einer **Zustandsübergangsfunktion** $next : State \times In \to State$, einer **Ausgabefunktion** $out : State \times In \to Out^*$ und einem **Anfangszustand** $q_0 \in$ **State** .

blaaaaa..... kapier ich nicht. Kann jemand anderes mir den Müll erklären?

8.3 Fallstudie: Interaktiver Keller

9 Zustands- und Ablauf orientierte Spezifikation

9.1 Funktionale (semantische) Softwarequalität

Was bedeutet das? Es ist im Prinzip wieder eine Modellierungsart die uns schon häufiger über den Weg gelaufen ist. Dabei geht es darum Systeme so abstrakt wie möglich und so genau wie nötig zu modellieren. Es geht zusätzlich darum die Eigenschaften der Funktionen eines Systems möglichst genau zu spezifizieren, also zu beschreiben. Das Modell des Systems sollte dann mit den Eigenschaften direkt vergleichbar sein.

9.1.1 Anwendungsbeispiele:

Formale Methoden zur Modellierung von Systemen und Spezifizierung der Funktionen eines solchen ist die Prädikatenlogik, Automaten(Transitionssysteme wie NFA, DFA etc.) und die Temporallogik.

9.2 Transpositionssysteme

Genau das selbe wie ein DFA oder NFA nur das es keine akzeptierende Zustände gibt, da es nur um die Übergänge geht. Diese Automaten Art hat unendlich viele Zustände und beginnt mit dem Zustand s. Zustand...

9.3 Temporallogik

9.3.1 CTL*: Computition-Tree(Star) Logic

Bei der **Temporallogik** geht es darum zeitliche Zusammenhänge zu beschreiben. Mit einer Ähnlichen Syntax und Semantik wie in der Aussagenlogik wird beschrieben wann und ob ein bestimmter Zustand erreicht wird. CTL* ist der Überbegriff für die zwei Teillogiken CTL und LTL(linear-time temporal logic).

LTL

Unäre Operatoren:

- Xφ φ gilt an der nächsten Position

- Gφ φ gilt auf dem kompletten nachfolgenden Pfad
- Fφ φ gilt irgendwann auf dem nachfolgenden Pfad

Binäre Operatoren:

- ψ Uφ Irgendwann gilt φ. Bis dahin gilt ψ.
- ψ Rφ φ gilt(einschließlich) bis zu der Position, an der ψ gilt.

Hierbei sind φ und ψ irgendwelche LTL-Formeln.

Beispiele:
Irgendwann gilt φ und überall gilt ψ! : $F\varphi \wedge G\psi$
Ab der übernächsten Position gilt immer φ! : $XXG\varphi$
Es gilt fast immer φ! : $FG\varphi$

CTL

Das selbe wie LTL nur das mehrere Pfade möglich sind. Es gilt:

- $E\varphi$ Es gibt einen Pfad, auf dem φ gilt.
- $A\varphi$ Auf allen Pfaden gilt φ

Beispiele:

- $EG\varphi$: Es gibt einen Pfad, auf dem überall φ gilt
- $AX\varphi$: In jedem nächsten Zustand gilt φ
- $\not{E}F\ \not{\varphi}$: Es gibt keinen Pfad, auf dem φ irgendwann nicht gilt.

10 Abstraktionsebenen von Datenstrukturen

Es gibt Maschinennahe Datenstrukturen (Beispiel: Arrays Binärworte, Files auf einem Datenband) nun stellt sich natürlich die Frage wie man andere Dinge in Datenstrukturen abbildet (bsp. Graphen, Mengen, Tabellen, Sequenzen).
Hierbei sieht man sich Mengen, Elemente und Funktionen an und unterscheidet hierbei zwischen externem Verhalten und internen Realisierungen. Beispiel für unterschiedlichen internen Realisierungen: Dezimalzahlen, Binärzahlen, römischen Ziffern und Strichdarstellung, extern haben aber alle das selbe Verhalten.

10.1 Algebraische Darstellung

Bei der algebraischen Darstellung von abstrakten Datentypen werden die Datentypen und ihre Operationen axiomatisch dargestellt.
In der algebraischen Darstellung hat jeder Typ eine Signatur welche die Schnittstelle einer Datenstruktur darstellt, indem sie Sorten, Konstantsymbole und Operationssymbole spezifiziert.
Semantisch kann man sagen, dass die Sorten die verwendeten Mengen sind (z.B. natürliche Zahlen und Booleans, praktisch gesehen könnte man Sagen die Datentypen die ïmportiert"werden müssen), die Konstantensymbole die Elemente (z.b. bei natürlichen Zahlen 1,2,3,4,5...). und die Operationen (partielle) Funktionen.
Eine Signatur wird wie folgt angegeben:
$\Sigma = (\mathbb{S}, \mathbb{K}, \mathbb{F})$ Und im angewendeten Fall:
$\mathsf{A} = (\mathbb{S}(\mathsf{A}), \mathbb{K}(\mathsf{A}), \mathbb{F}(\mathsf{A}))$
Man gibt eine Algebra in der Regel in einer Art Tabelle in in der unter spec der neue Typ steht, unter
sorts die verwendeten Typen, unter
ops die Operationen und ihre Parameter (in der folgender Form: $(EingabeTyp_1, EingabeTyp_2, ..., EingabeTy$
und,
Axioms: die Regeln die für die Operationen gelten.

Beispiel aus der Vorlesung: Wahrheitswerte
spec Bool ; Der neue Datentyp
sorts bool = true | false ; die verwendeten Datentypen
ops not = (bool)bool ; Eine Operation ihre Parameter und Rückgabewerte

vars x:bool ;Variablen die in den Axiomen vorkommen
axioms $true \neq false$;
not(true) = false
not(false) = true

Anders Spezifiziert nun wäre dies
$\mathsf{A} = (\mathbb{S}^{\mathsf{A}}, \mathbb{K}^{\mathsf{A}}, \mathbb{F}^{\mathsf{A}})$
$\mathbb{S} = bool$
$\mathbb{K}^{bool} = true, false$
$\mathbb{F}^{(bool)bool} = not$

To Be Done : Basis und Familie, keine Ahnung was das ist.

11 Objektorientierte Programmierung

Da jeder der sich mit OOP nicht auskennt hart gefailt hat im ersten Semester setz ich eigentlich voraus, dass das jeder kann, daher hier nur wichtige Schlagworte die man können muss:

11.1 Stichworte

1. Objekt und Klasse
2. Attributwerte (Objekteigene Variablen)
3. Operationen (Methoden)
4. Kapselung (Information Hiding) Private,Public,package private, protected
5. Klassenatribute (static)
6. Klassenoperationen (static methoden)
7. Klassifikation von Operationen (constructor, setter (update), getter (access),destructor)
8. Polymorphie (Abstrakte Klassen, Interfaces, Vererbung)

12 UML

UML (Unified Modeling Language) ist der Industriestandard der die Graphische Notation von objektorientierten Modellierungen ermöglicht. UML's sind Diagramme zur Modellierung von Systemen und ind unabhängig von Programmiersprachen (sind aber objektorientiert).
UML bietet Möglichkeiten, die Strukturelle Sicht und die Verhaltenssicht zu Modellieren.

12.1 Strukturelle Sicht

Die strukturelle Sicht besteht aus Diagrammen für Klassen, Pakete, Obejkte, Verteilungen (Verteilung auf Hardwareeinheiten) und Komponenten und Kompositionsstrukturen.

12.2 Verhaltenssicht

Die Verhaltenssicht bietet Anwendungsfalldiagramme (Benutzersicht aufs System), Sequenzdiagramme (Interaktion von Objekten) Zustandsdiagramme (zustandsbasierte Verhalten), Aktivitätendiagramme (Systemabläufe), Kommunikations und Zeiterverlaufsdiagramme.

12.3 LEST DAS!

Die Wikipedia seite zu UML liefert hier jede Info besser als ich es jemals könnte. Daher: http://de.wikipedia.org/wiki/UML .

13 Validierung und Test

Validierung: Entspricht das Produkt den Kundenwünschen.
Verifikation: Bauen wir das Produkt auf die richtige Weise.
Softwareinspektion ist ein statisches Prüfungsverfahren das in jeder Phase angewendet werden kann und die Systembeschreibung analyisert.
Softwaretests eigentlich selbsterklärend sind tests des Systems mit eingabewerten und Analyse der Ausgaben und des Systemverhaltens. Dies kann selbstverständlich nur bei fertigen Modulen oder Prototypen gemacht werden.
Es gibt zwei zu testende Verhalten, das **funktionale Verhalten** (Fehler im Programm) und das Ausführungsverhalten (stress test, Ausführungsverhalten des Programms, Ausführzeiten, Datendurchsatz usw.). Wie aufwändig die durchzuführenden Tests sind ist abhängig von dem Einsatzgebiet (Kritische Prozessführung?), Benutzerwartungen, Marktlage und Lebensdauer des Systems. Hierbei ist festzuhalten, dass kein größeres System fehlerfrei ist - NIE NIE NIEMALS.
Debugging ist der Vorgang bei dem Fehler lokalisiert und behoben werden.

13.1 Softwareinspektion

Softwareinspektion untersucht die Systembeschreibung auf systematische Fehler. Gegenüber dem Test hat es einige Vorteile: Geringerer Zeitaufwand, in jeder Phase anwendbar, Programmierwissen kann eingebracht werden und Verbesserungen in Entwurf und Struktur sind möglich (und nicht erst - wie bei Tests - wenn das Modul fertig ist).

13.1.1 Rollen bei der Inspektion von Code

Es gibt prinzipiell fünf Rollen: Den Inspektor (findet fehler) den Leser (erklärt den Code) den Schriftführer (Protokollant), den Autor (behebt die Fehler) und den Vorsitzenden (Moderator).

13.1.2 Fehlerklassen

Fehler werden in verschiedene Kathegorien die relativ selbsterklärend sind gesteckt.

1. Datenstrukturfehler

2. Kontrollstrukturfehler
3. Ein-/Ausgabefehler
4. Schnittstellenfehler
5. Speicherverwaltungsfehler
6. Ausnahmebehandlungsfehler
7. Kommunikationsfehler
8. Fehler im Zeitverhalten

13.1.3 Automatische statische Analyse

Die automatische (statische) Analyse analysiert den Quellcode zur Compilierzeit (oder schon vorher, siehe Eclipse) auf mögliche Fehler. Probleme die hierbei aufgedeckt werden könne sind:
Datenflussfehler (falsche Typisierungen, nicht initialisierte Variablen), Kontrollflussfehler (z.b. nach einem return oder break Statement noch code) und Schnittstellenanalysen (Versuch auf private Variablen zuzugreifen, inkompaktible Schnittstellen (integer in float speichern..))

13.2 Softwaretests

Softwaretests sind systematische Verfahren um Bugs zu finden.

13.2.1 Defekttest

Hier werden verschiedene Systemzustände getestet und das System mit Testdaten gefüttert, das Ergebis anschließend analysiert und ein Bericht erstellt.

Black-Box-Test

Hier werden Daten eingegeben und die Ausgaben angeguckt, ins System wird hierbei nicht geschaut, daher „Black-Box“-Test. Die Eingabedaten werden hierbei in Eingabeäquivalenzklassen partitioniert und die Testfälle aus diesen gewählt (Ein bisschen wie Nerode Klassen).

13.2.2 Integrationstesten

Beim Integrationstesten wird die Zusammenarbeit von bereits getesteten Komponenten überprüft.

14 Literaturverzeichnis

[Her09] Herczeg, Michael: *Software-Ergonomie: Theorien, Modelle und Kriterien für gebrauchstaugliche interaktive Computersysteme*. Oldenbourg Wissenschaftsverlag, vollständig überarbeitete und erweiterte auflage Auflage, März 2009, ISBN 9783486587258. http://amazon.de/o/ASIN/3486587250/.