Fachhochschule
Kaiserslautern
University of Applied Sciences

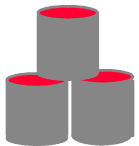
Informatik und
Mikrosystemtechnik
Zweibrücken

Grundlagen der Informatik
– Software-Entwicklung –

Prof. Dr. Bernhard Schiefer

(basierend auf Unterlagen von Prof. Dr. Duque-Antón)

bernhard.schiefer@fh-kl.de
http://www.fh-kl.de/~schiefer



Inhalt

- Software-Entwicklung

Software-Entwicklung

- Entwicklung von Software ist ein hoch komplizierter Prozess. Umso umfangreicher die Software, umso problematischer auch die Entwicklung.
- Wie kann die Komplexität (Umfang) eines Software-Programms gemessen werden? Es gibt kein exaktes Maß dafür., einige Beispiele sind:
  - ⇨ Anzahl der Quelltextzeilen des Programms, Line of Codes (LOC).
  - ⇨ Entwicklungszeit, die zur Erstellung des Programms benötigt wurde, z.B. Mann-Jahre.
- Software-Komplexität ist keine exakte Wissenschaft, beide Maße sind offensichtlich nicht sehr genau.

Software-Entwicklung

- Die folgende Tabelle gibt einen groben Einblick in die Klassifikation von Software-Projekten hinsichtlich ihrer Komplexität

Software-Klasse	Codezeilen (LOC)	Bearbeitungsaufwand (PJ)
Sehr klein	0 – 1.000	0 - 0,2
Klein	1.000 – 10.000	0,2 - 2
Mittel	10.000 – 100.000	2 - 20
Groß	100.000 – 1.000.000	20 - 200
Sehr groß	1 Mio. - …	200 - …

LOC – Lines of code
PJ – Personenjahre

Hauptprobleme bei der Software-Entwicklung

- Ein Hauptproblem der Software-Entwicklung ist die Zuverlässigkeit bzw. Fehlerfreiheit:
  - ⇨ Zuverlässigkeit: Programm verhält sich im Betrieb so wie es soll (wie in den Spezifikationen beschrieben)
  - ⇨ Je umfangreicher das Programm, desto unwahrscheinlicher ist es, dass es sich zuverlässig verhält und immer fehlerfreie Ergebnisse/Abläufe garantiert
- Man geht davon aus, dass es im allgemeinen unmöglich ist, umfangreiche Software-Produkte vollständig fehlerfrei zu entwickeln

Hauptprobleme bei der Software-Entwicklung

- Ein weiteres Problem besteht darin, die Software so zu entwickeln, dass spätere (noch unbekannte) Änderungen eingefügt werden können
  - ⇨ Konkreter Entwickler möchte sein Programm so schnell wie möglich beenden
  - ⇨ Software-Entwicklung ist jedoch ein evolutionärer Prozess, der oft sehr lange andauert, und dessen Ende womöglich noch nicht abzusehen ist
  - ⇨ Schwer zu entscheiden, wie viel Programmieraufwand (Zeit & Kosten) in die Entwicklung der „Ausbaufähigkeit“ gesteckt werden soll
    - zu Beginn ist noch unklar ob/wie oft die Software in Zukunft eingesetzt werden wird (und Umsatz bringt).
  - ⇨ Was sind die Gründe für notwendige Software-Änderungen?

Gründe für spätere Software-Änderungen

- es werden Fehler entdeckt
  - ⇨ diese müssen für einen begrenzten Zeitraum in der Regel kostenlos entfernt werden
- es werden neue Anforderungen an die Software gestellt
  - ⇨ z.B. bei Änderung gesetzlicher, betrieblicher oder organisatorischer Rahmenbedingungen
    - Änderung in der Gesetzgebung implizieren immer wieder Anpassungen in entsprechenden Software
  - ⇨ Für diese können, je nach Wartungsvertrag, separat Kosten anfallen
- Software soll/muss in neuer Hardware-Umgebung eingesetzt werden
- Software will/muss zusätzliche Hardware-Komponenten nutzen
  - ⇨ weil z.B. über neue Schnittstellen neue Geräte angeschlossen werden sollen

Lösungen: Methoden und Werkzeuge

- Seit Ende der 60er beschäftigt man sich in der Informatik mit der Frage, wie die Entwicklung von qualitativ hochwertiger Software ablaufen sollte → Software Engineering
- Wichtige Kriterien für die Qualität solcher Software sind:
  - ⇨ Zuverlässigkeit und Korrektheit (also Fehlerfreiheit)
  - ⇨ Modifizierbarkeit, Wartbarkeit, Testbarkeit und Wiederverwendbarkeit
  - ⇨ Effizienz
  - ⇨ Kosten
- Eine ideale Software-Lösung, die völlig fehlerfrei, kostengünstig ist und die man später leicht ändern kann, gibt es nicht.

Lösungen: Methoden und Werkzeuge

- Wie kann das Ziel, qualitative möglichst gute Software zu schreiben, systematisch unterstützt werden?
  - ⇨ Durch ein systematisches Vorgehen im Projekt.
  - ⇨ Durch die geeignete Berücksichtigung entsprechender Faktoren zur Produktivität.
  - ⇨ Durch die Einführung geeigneter Methoden zur Software-Entwicklung.

Vorgehensmodelle

- Die entsprechenden Vorgehensmodelle wurden bereits in Kapitel „Grundlagen der Programmierung“ beschrieben.
- Im wesentlichen unterscheidet man die Phasen:
  - ⇨ Sammeln der Anforderungen
  - ⇨ Entwurf
  - ⇨ Codierung
  - ⇨ Test und Integration (Modul-Test und Gesamt-Test)
- Alle Modelle sind iterativ und bieten die Möglichkeit des Feedbacks, so dass beispielsweise in der Phase der Codierung ein Fehler im Entwurf festgestellt wird und daher wieder zu Entwurfs-Phase gesprungen werden muss.
- Die einzelnen Modelle unterscheiden sich in der Frage, wie innerhalb der Phasen gesprungen werden darf.

Produktivität

- Für die Produktivität sind viele Faktoren von Bedeutung. Dazu gehören:
  - ⇨ Ausstattung mit leistungsstarken Rechner und Bereitstellung von ausreichenden Netzverbindungen.
  - ⇨ Ausstattung mit geeigneten Programmierumgebungen, auch Software-Entwicklungs-Umgebungen genannt.
  - ⇨ Peopleware
    - Mitarbeiterführung im Sinne von Menschenführung
    - Auswahl geeigneter Mitarbeiter
    - Förderung der Teambildung
    - Angemessene Gestaltung der Büroumgebung
    - Erhaltung der *Work-Life-Balance*

Methoden der Software-Entwicklung

- Die traditionellen Methoden zur Programmentwicklung wurden von Dijkstra, Hoare, Wirth, Mills, Parna und viele anderen bereits ab 1970 propagiert. Sie sind so grundlegend und allgemein gültig, dass sie sich auch in den späteren Methoden und Ansätzen wiederfinden:
  - ⇨ Strukturierte Programmierung
  - ⇨ Schrittweise Verfeinerung und Top-Down-Entwurf.
- Die Methoden der ersten Generation sind für größere Systeme nicht mehr gut geeignet.
- Ab Ende der 70er Jahre wurde daher einen neue, zweite Generation von Methoden zunehmend industriell eingesetzt:
  - ⇨ Modularisierung von Software nach dem Daten-Abstraktionsprinzip zusätzlich zur bisherigen funktionsorientierten Entwicklung.

Neue Methode: Objektorientierung

- Seit Beginn der 80er Jahre hat sich die Objektorientierung in der Softwaretechnik kontinuierlich ausgebreitet und ist heute beherrschend.
  - ⇨ Dahinter verbirgt sich zunächst einmal eine neue Denkweise in der Programmierung
  - ⇨ Die Anwendung wird als Menge miteinander agierender Objekte modelliert

Strukturierte Programmierung

- Wesentliches Ziel der strukturierten Programmierung:
  - ⇨ Verbesserung von Lesbarkeit und auch Verbesserung der Korrektheit von Software.
- Dazu werden u.a. die folgenden Regeln empfohlen:
  - ⇨ Verwende nur strukturierte Anweisungen wie Fallunterscheidung- und Schleifen-Anweisungen, aber keine goto-Anweisungen.
  - ⇨ Baue das Programm so auf, dass einzelne Teile leicht abtrennbar (und damit separat leicht verifizierbar) sind, z.B. durch ausgiebige Verwendungen von Prozeduren/Funktionen (Methoden).
  - ⇨ Verwende klar aufgebaute (und separat vereinbarte) Datenstrukturen und Konstantenvereinbarungen statt über das Programm verteilte Codeschlüssel.
  - ⇨ Verwende selbsterklärende und programmspezifische Bezeichner.

Strukturierte Programmierung

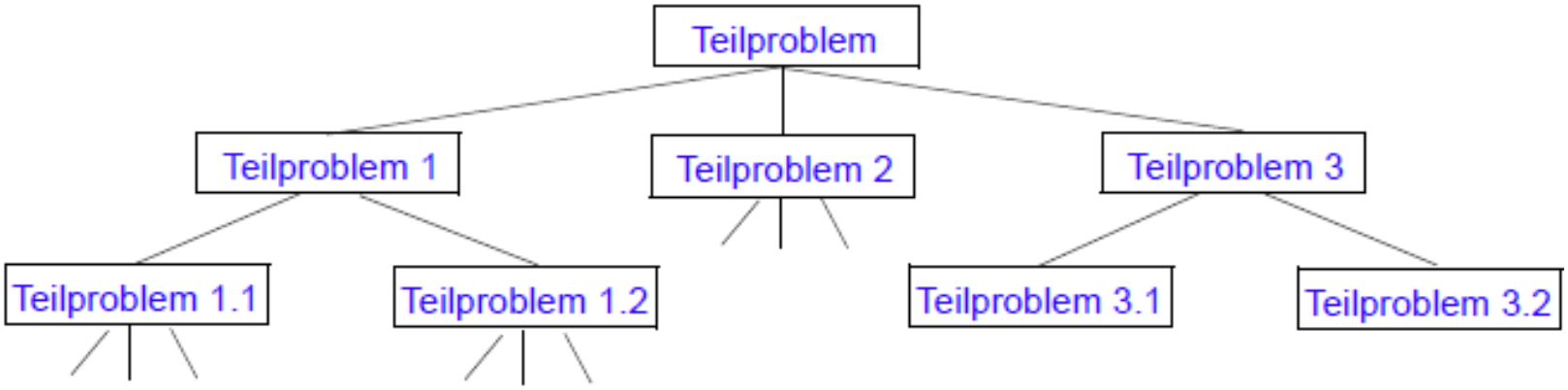
- Aktuelle Programmiersprachen unterstützen durch die angebotenen bzw. weggelassenen Sprachelemente das strukturierte Programmieren.
- In erster Linie hängt es aber von der Fertigkeiten und der Disziplin des Programmierers ab, wie gut seine Programme strukturiert sind.
- Des weiteren gibt es in den meisten Firmen bindende Richtlinien zur Programmierung, die in der Regel durch Werkzeuge unterstützt werden und somit die Einhaltung der Regeln erzwingen.

Schrittweise Verfeinerung und Top-Down Entwurf

- Die strukturierte Programmierung gibt Regeln für das „Programmieren im Kleinen" an:
  - ⇨ Sie beschreibt die innere Gestaltung kleinerer Programme.
- Für die Erstellung mittlerer bis großer Programmsysteme wurde das Konzept der *schrittweisen Verfeinerung* entworfen, das bereits im Kapitel „Algorithmen und Techniken“ aus algorithmischer Sicht erläutert wurde.
  - ⇨ Aus großen, schwer überschaubaren Problembereichen werden kleinere Teilprobleme herausgelöst und gesondert bearbeitet.
  - ⇨ Dieser Schritt kann mehrfach wiederholt werden und hat die schrittweise Verfeinerung ganzer Programmteile zur Folge

Grafische Darstellung: Top-Down-Entwurf

- Die grafische Darstellung dieses Prozesses liefert einen Baum:



Daten- und Funktionsorientierte Methoden

- Methoden der ersten Generation wie die strukturierte Programmierung reichen als methodischer Wegweiser nicht mehr aus, wenn es um die Probleme des „Programmierens im Großen" geht. Also wenn
  - ⇨ man große Programmieraufgaben in separat zu behandelnde Teilaufgaben zerlegen will und
  - ⇨ dabei Daten und Funktionen gemeinsam in geeigneter Weise strukturieren will.
- Die zweite Generation von Methoden hat dazu Ende der 70er zeitgleich verschiedene Ansätze vorangetrieben:
  - ⇨ Parnas hat das „*Geheimnisprinzip*" mit Hilfe der Daten-Abstraktion und der Modularisierung eingeführt.
  - ⇨ Die strukturierte Programmierung wurde durch geeignete Hilfsmittel (Datenfluss-Diagramm) ausgebaut zur Strukturierten Analyse.
  - ⇨ Aus dem Datenbank-Entwurf wurde die Entity/Relationship-Modellierung nach kurzer Zeit auch in der Software-Technik übernommen.

Maßnahmen zur Qualitätssicherung

- Unabhängig von der Einführung des Konzepts zur Datenstrukturierung wurden auf Grund der hohen Komplexität der zu lösenden Software-Projekte Maßnahmen zur Qualitätssicherung notwendig:
  - ⇨ Systematische Test-, Review und Inspektionsverfahren wurden daher zunehmend etabliert.

Information Hiding (Geheimnisprinzip)

- Bei der Entwicklung großer Software-Systeme stellt die **Kommunikation zwischen den Entwicklern** eines der größten Probleme dar:
  - ⇨ Jede Person bearbeitet einzelne Programmbausteine oder auch Module genannt
    - Verknüpft sind diese miteinander sowohl über Kontroll- als auch Datenstrukturen
  - ⇨ Diese Module müssen so definiert und gegeneinander abgegrenzt werden, dass jeder Kollege (Co-Entwickler) das für ihn Notwendige über diese erfährt.
    - Aber nicht mit Detailwissen über die Module der anderen überfrachtet wird

Information Hiding (Geheimnisprinzip)

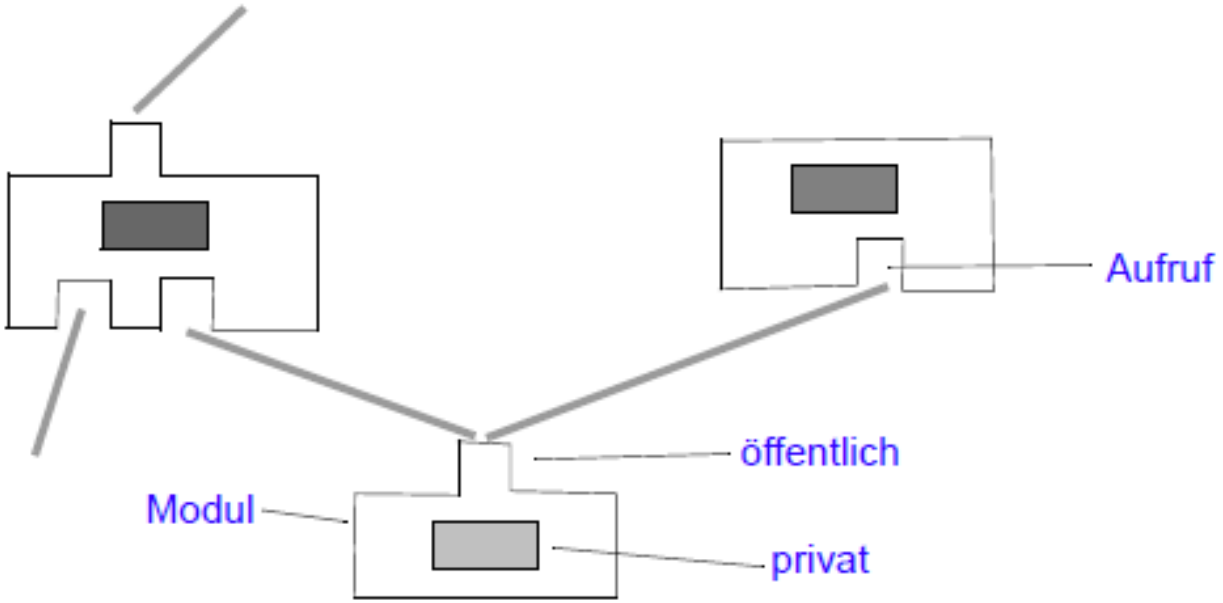
- Parnas schlug 1972 dazu das *Geheimnisprinzip* vor, welches besagt, dass beim Entwurf eines großen Systems jedes Modul in zwei Teilen zu beschreiben ist:
  - ⇨ Alle Vereinbarungen, die für die Benutzung des Moduls durch andere Module notwendig sind.
    - Der sogenannte „Visible Part" oder auch Spezifikation genannt.
  - ⇨ Alle Vereinbarungen und Anweisungen, die für die Benutzung des Moduls durch andere Module nicht benötigt werden.
    - Der sogenannte „Private Part" oder auch Konstruktion genannt.
- Das war der Grundstein für das wenige Jahre später definierte Prinzip der Datenabstraktion.

Prinzip der Datenabstraktion/Datenkapselung

- Daten und darauf operierende Funktionen (Operationen) müssen immer gemeinsam in einem unmittelbaren Zusammenhang definiert werden.
- Datenstrukturen sind so in Module zu verpacken (verkapseln), dass auf sie von anderen Modulen nur über ihre Operationen zugegriffen werden kann.
- Die Beschreibung (Schnittstelle bzw. Signatur) dieser Operationen macht die Spezifikation des Moduls aus, die für alle anderen sichtbar ist.
- Die Programmierung der konkreten Datenzugriffe erfolgt im privaten Konstruktionsteil
  - ⇨ und damit im alleinigen Verantwortungsbereichs des damit befassten Entwicklers

Grafische Darstellung: Datenkapselung

- Diese Abbildung beschreibt den Aufbau eines Software-Systems nach dem Prinzip der Datenkapselung (Data Encapsulation), was zu einer Modularisierungsstruktur führt

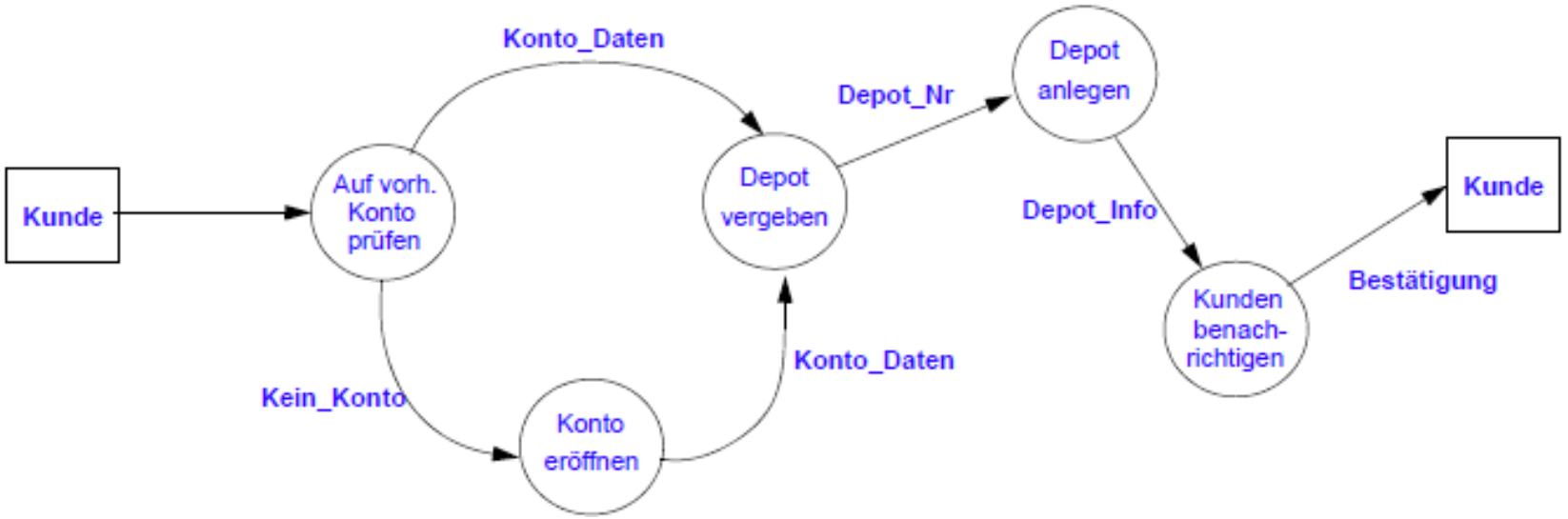


Strukturierte Analyse

- Im Vordergrund stehen grafische Dokumentationsformen.
  Ein Datenfluss-Diagramm (data flow diagram) ist ein einfacher gerichteter Graph,
  - ⇨ dessen Knoten Tätigkeiten, Aktionen oder Vorgänge und
  - ⇨ dessen Kanten für zwischen diesen transferierte Informationseinheiten stehen, den sogenannten Datenflüssen.
  - ⇨ Spezialsymbole für interne Datenspeicher und für externe Quellen und Senken von Informationen vervollständigen das grafische Vokabular.

Beispiel: Datenflussdiagramm

- Beispiel zeigt ein einfaches Datenflussdiagramm für die Depoteinrichtung bei einer Bank:



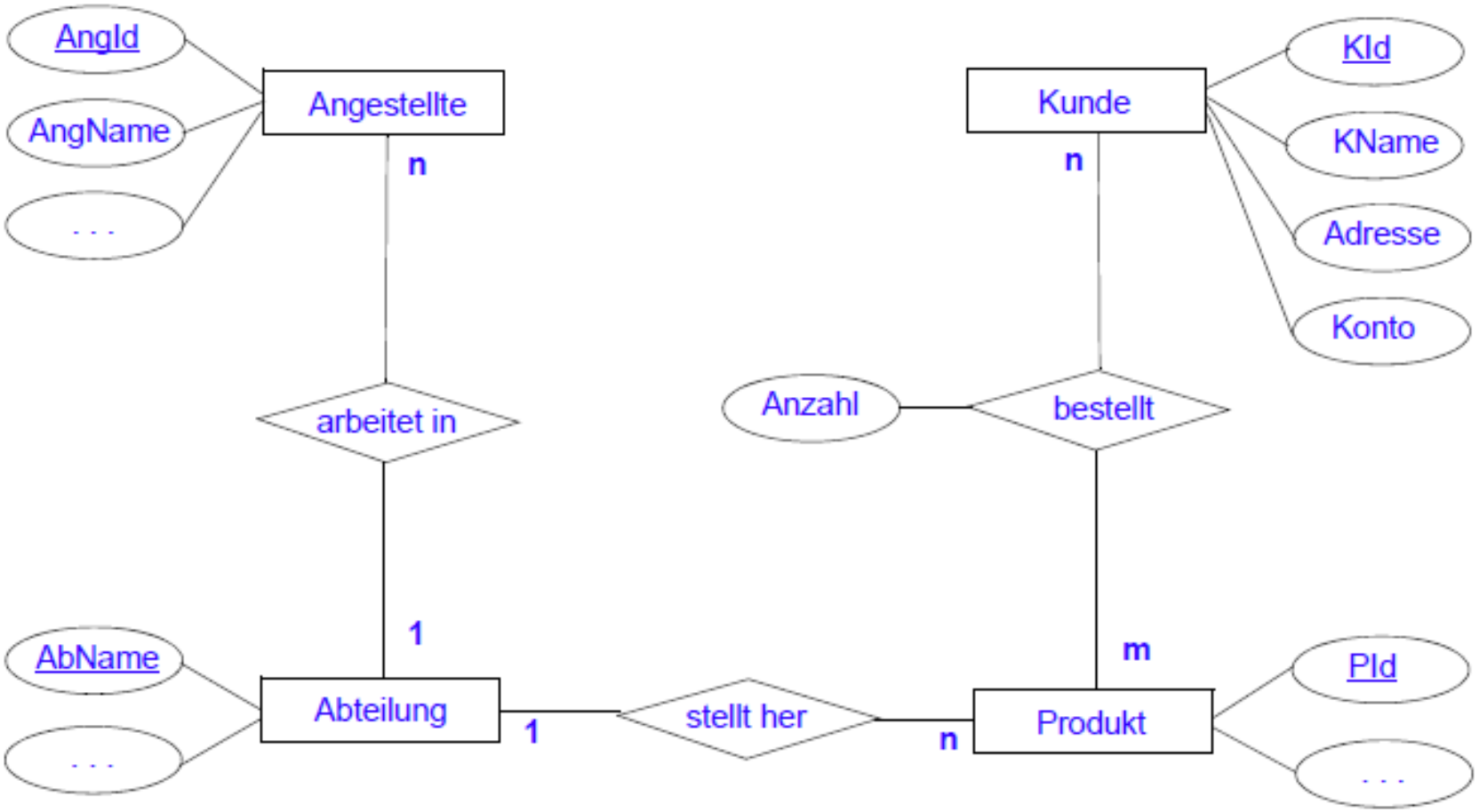
Entity/Relationship-Modellierung

- 1976 veröffentlichte Peter Chen seine Arbeit über das E/R-Modell
  - ⇨ Gedacht als Entwurfstechnik für komplexe Datenbank-Strukturen
- Bei der Entwicklung größerer Software-Systeme wird häufig die Datenmodellierung als erster Entwurfsschritt vorangestellt
  - ⇨ Die Datenmodellierung erlaubt die Konzentration auf die konzeptionellen Zusammenhänge der zu speichernden Informationen und zwar unabhängig von einer späteren Implementierung.

Entity/Relationship-Modellierung

- Die E/R-Technik ermöglicht beim Entwurf großer Software-System, zunächst einmal die Abstraktion von funktionalen Zusammenhängen und damit von der Systemdynamik.
  - ⇨ Natürlich ist damit keine vollständige Systembeschreibung zu erreichen. Doch lasen sich E/R-Diagramme relativ leicht mit Darstellungen funktionaler Zusammenhänge wie z.B. Datenfluss-Diagrammen kombinieren.
- Ein E/R-Datenmodell lässt sich als Graph (E/R-Diagramm) veranschaulichen.
  - ⇨ Die Knoten werden als Rechtecke oder Rauten dargestellt, je nachdem ob sie Entitäten oder Beziehungen darstellen.
  - ⇨ Falls ein Entitätstyp im entsprechenden Beziehungstyp enthalten ist, wird zwischen den beiden Knoten eine Kante gezogen.
  - ⇨ Die Kante besitzt als Markierung die Kardinalität, die der Entitätstyp in dem Beziehungstyp besitzt.

E/R-Beispiel: Mini-Unternehmen



Objektorientierte Software-Entwicklungsmethoden

- Seit der Programmiersprache Smalltalk zu Beginn der 80er Jahre hat sich der Begriff „objektorientiert“ in der Softwaretechnik kontinuierlich verbreitet und ist heute beherrschend.
- Insbesondere werden durch die OO-Analyse die Inhalte der herkömmlichen Daten- und Funktionsmodelle zusammengefasst.
- Zur Beschreibung der OO-Modellierung über diese Phasen hinweg wird die Modellierungssprache UML (Unified Modelling Language) eingesetzt:
  - ⇨ UML ist grafikbasiert und bietet verschiedene Diagramm-Typen zur Darstellung der verschiedenen Aspekte an.
  - ⇨ UML hat sich als Standard zur einheitlichen Modellierung eines OO-Systems etabliert.

UML: Unified Modelling Language

- UML ist eine sehr mächtige Sprache zum Beschreiben objektorientierter Softwaremodelle
  - ⇨ Wird genutzt zur Spezifikation, Visualisierung, Konstruktion und Dokumentation der Bestandteile eines Softwaresystems
- UML besteht aus drei Hauptelementen:
  - ⇨ Grundbestandteile
  - ⇨ Regeln zum Zusammensetzen
  - ⇨ Allgemeine Mechanismen
- UML ist keine visuelle Programmiersprache, jedoch lassen sich ihre Modelle in vielen Programmiersprachen abbilden

Diagrammtypen der UML

- UML hat verschiedene Diagrammtypen, um ein System mit mehreren Schichten darzustellen:
  - ⇨ Klassendiagramm
    - Zeigt eine Menge von Klassen und ihre Beziehungen
  - ⇨ Objektdiagramm
    - Zeigt eine Menge von Objekten und ihre Beziehungen
  - ⇨ Sequenzdiagramm
  - ⇨ Kollaborationsdiagramm
  - ⇨ Zustandsdiagramm
  - ⇨ ….-diagramm

Prinzipien der Objektorientierung

- Prinzipiell kann jede speicherbare Größe als Objekt aufgefasst werden,
  - ⇨ das nicht nur passiven Charakter hat wie eine herkömmliche Variable,
  - ⇨ sondern zugleich aktiv werden kann durch eigene Operationen (in Java Methoden genannt), die z.B. Nachrichten an andere Objekte versenden oder selbst welche empfangen und darauf reagieren können.
- Merkmale eines Objektes sind:
  - ⇨ Attribute (Variablen)
  - ⇨ Methoden (Operationen, welche Dynamik beschreiben)
- Weitere wichtige Konzepte der OO-Technik:
  - ⇨ Vererbung
  - ⇨ Polymorphie
  - ⇨ Dynamisches Binden

Software-Qualitätssicherung

- Bei der Betrachtung der Methoden zur Software-Entwicklung stehen oft die frühen Phasen von der Analyse über den Entwurf zur Programmierung im Vordergrund.
- Ebenso wichtig (und oft noch kostenintensiver) sind die Tätigkeiten zur Überprüfung der Software in Bezug auf stabilen und zuverlässigen Einsatz.
- Unter Software-Qualitätssicherung (Software-QS) versteht man die Gesamtheit der konstruktiven und analytischen Maßnahmen, um die geforderte Qualität zu erhalten.
- Die Qualität eines Produkts (auch einer Software) kann nur in Relation zu geforderten Qualitätszielen oder Qualitätsanforderungen überprüft werden, möglichst in quantifizierbarer Form.

Software-Qualitätssicherung

- Daher beginnt der QS-Prozess zu Beginn des Projekts und nicht am Ende nur zur Überprüfung. Solche Anforderungen könnten beispielsweise sein:
  - ⇨ Die Antwortzeit darf im Normalbetrieb nicht länger als 2 Sekunden betragen, oder
  - ⇨ Der Speicherbedarf des Gesamtsystems darf im laufenden Betrieb 50 MB Hauptspeicher nicht übersteigen, oder
  - ⇨ Der Code ist zu 100% C1- getestet.
    C1 beschreibt ein Testüberdeckungsmaß
    (jeder Programmzweig wurde im Test mindestens einmal durchlaufen)
- Diese wenigen Beispiele zeigen bereits die große Bandbreite von Qualitätskriterien.

Software-Qualitätssicherung: Maßnahmen

- Kategorien für mögliche Maßnahmen zur Qualitätssicherung :
  - ⇨ konstruktive Maßnahmen
  - ⇨ analytische Maßnahmen
- Konstruktive Maßnahmen:
  - ⇨ Qualitätsplan aufstellen mit Qualitäts-Kriterien, entsprechender Relevanz und die daraus abgeleiteten Anforderungen.
  - ⇨ Zeitplan für Reviews, Inspektionen und sonstige QS-Prüfmaßnahmen erstellen.
  - ⇨ Richtlinien, Standards aber auch Muster für die zu erstellenden Ergebnisse verbreiten und sicherstellen, dass Entwickler diese auch verwenden.
  - ⇨ Entwicklungsprozess begleiten, dokumentieren und ggf. Qualitäts- und Zeitplan anpassen.

Software-Qualitätssicherung: Maßnahmen

- Analytische Maßnahmen:
  - ⇨ Reviews, Inspektionen und sonstige QS-Prüfmaßnahmen durchführen und dokumentieren.
  - ⇨ Aktionen, die als Ergebnis oben erwähnter Prüfmaßnahmen beschlossen wurden, in Gang setzen und verfolgen bzw. Einhaltung überprüfen.