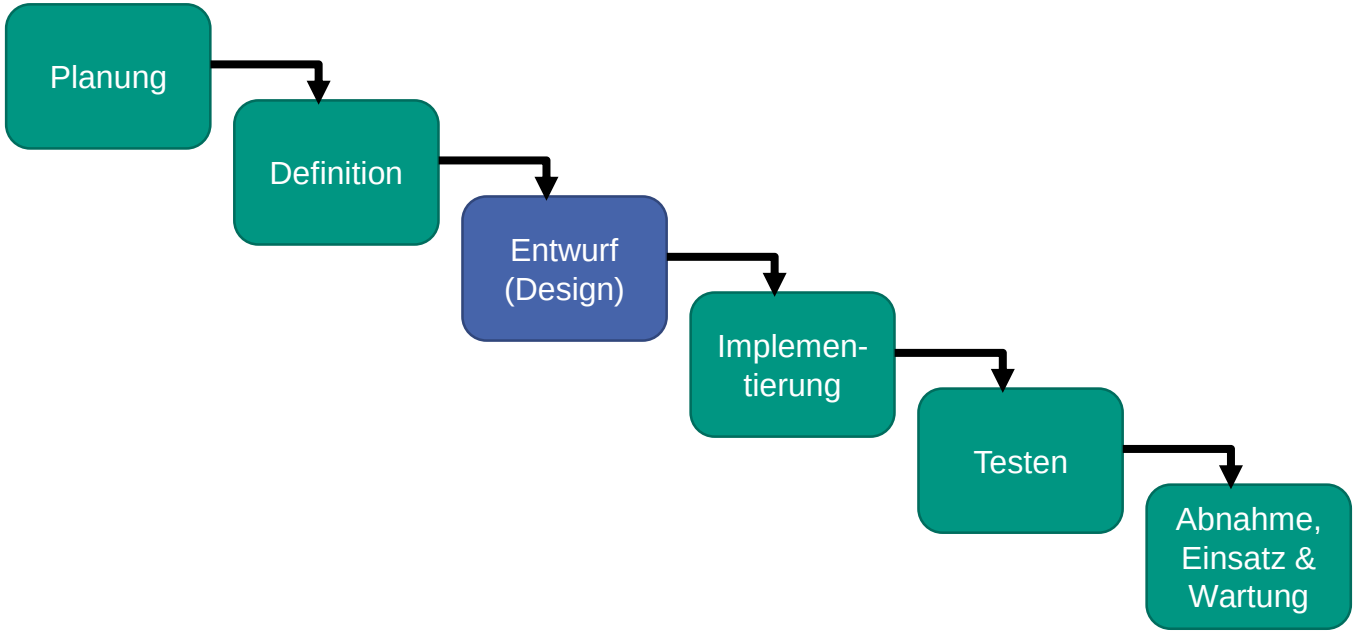
Karlsruher Institut für Technologie

Kapitel 3 – Die Entwurfsphase

SWT I – Sommersemester 2021

Walter F. Tichy, Christopher Gerking, Tobias Hey

Wo sind wir gerade?



Inhalt

- 3.1 Einführung und Überblick
- 3.2 Modularer Entwurf
  (siehe auch Balzert, LE 32)
- 3.3 Objekt-orientierter Entwurf
  (siehe Brügge, Dutoit, Kapitel 6)
- 3.4 Architekturmuster
  (siehe Brügge, Dutoit, Kapitel 6)
- 3.5 Entwurfsmuster
  (siehe Brügge, Dutoit, Kapitel 7)

3.1 Einführung und Überblick

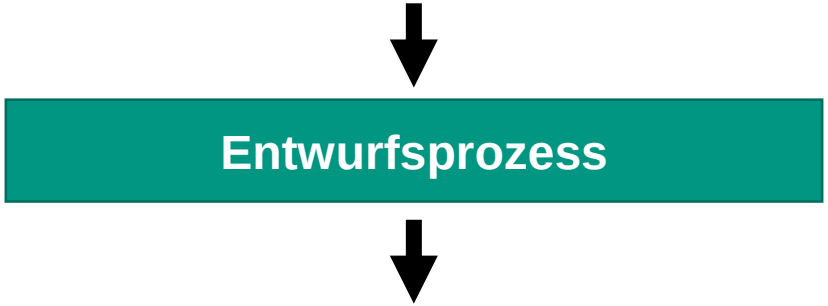
- Aufgabe des Entwerfens:
  - Ausgangspunkt: Anforderungen aus der Definitionsphase
  - Aufgabe: Aus den gegebenen Anforderungen an ein Software-Produkt eine software-technische Lösung im Sinne einer Softwarearchitektur entwickeln, die alle Anforderungen erfüllt.
  - Entwerfen wird auch »Programmieren im Großen« genannt.
  - Nach dem „Was bauen wir?“ der Definitionsphase folgt nun das „Wie strukturieren wir es?“ unter Berücksichtigung der Randbedingungen.

Softwarearchitektur

- Def. Softwarearchitektur (engl. *software architecture*):
  - Gliederung eines Softwaresystems in Komponenten (Module oder Klassen) und Subsysteme (Pakete, Bibliotheken). Dabei entsteht eine Bestandshierarchie
  - Spezifikation der Komponenten und Subsysteme
  - Aufstellung der »Benutzt«-Relation zwischen Komponenten und Subsystemen
  - Optional: Feinentwurf
    - Vorgabe der Datenstrukturen und Algorithmen, Pseudocode
  - Optional: Zuweisung von SW-Komponenten und Subsystemen zu HW-Einheiten
    - Bei verteilten Systemen, d.h. Systemen, die auf mehrere Rechner verteilt sind

3.1 Einführung und Überblick

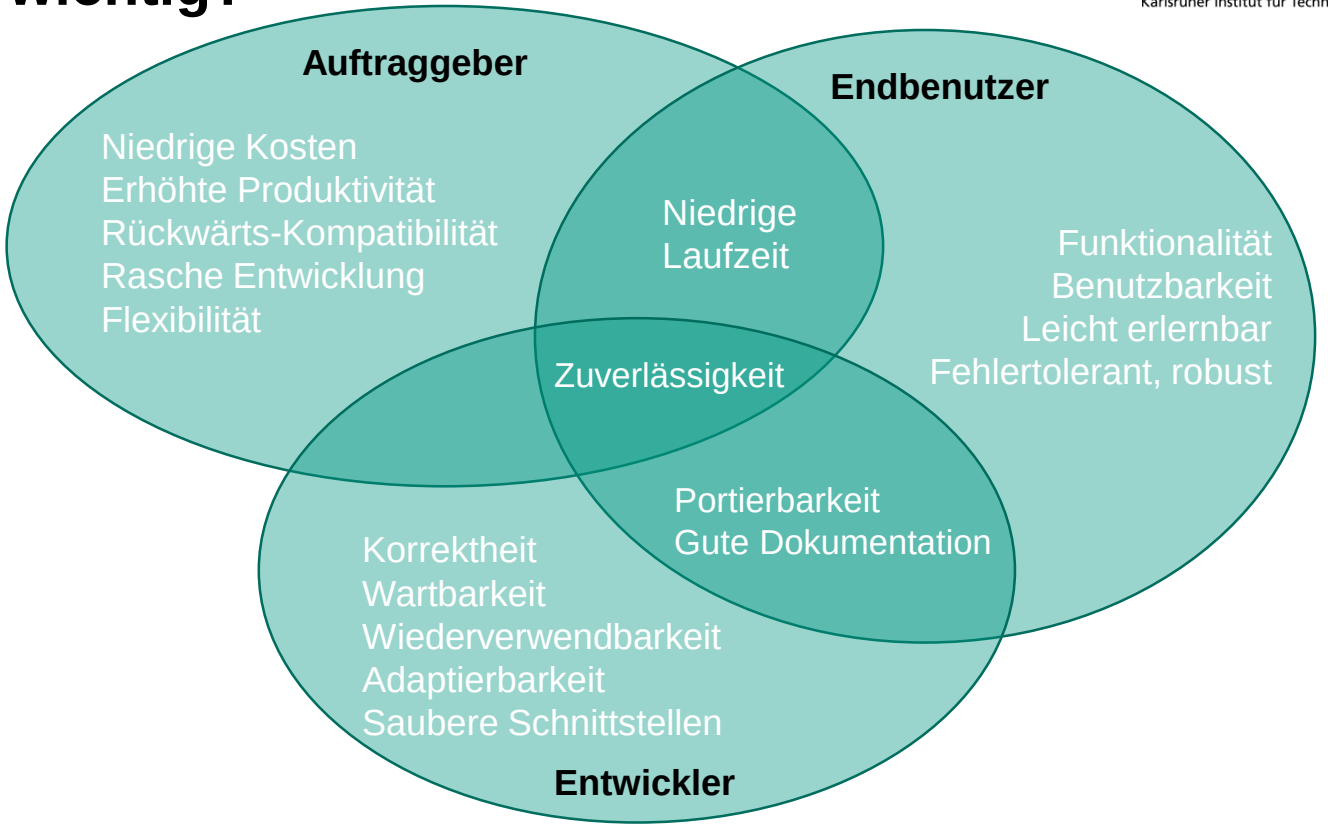
- Pflichtenheft (einschl. Modelle)
- Konzept Benutzungsoberfläche
- Benutzerhandbuch + Hilfekonzept



Weitere Fragen, die geklärt werden müssen

- Welche Komponenten existieren bereits und können wiederverwendet / gekauft werden, oder sind frei erhältlich?
  - In diesem Fall muss die Architektur ggf. angepasst werden!
- Liegen Echtzeitbedingungen vor, und wie werden sie eingehalten?
- Wie werden persistente Daten dauerhaft gespeichert?
  - In Dateien oder in Datenbanken?
- Wie ist die Ablaufsteuerung organisiert?
  - Monolithisch (über Aufrufe), ereignisgetrieben, parallel?
- Wie wird das System installiert, gestartet und angehalten?
- Wie werden Ausnahmen/Fehler behandelt?
- Wie werden Zugriffsrechte vergeben und überwacht?

Nichtfunktionale Anforderungen -- Wem sind sie wichtig?



Typische Entwurfs-Abwägungen

- Nur eine Untermenge der nichtfunktionalen Anforderungen kann gleichzeitig erfüllt werden.
- Ansonsten muss abgewogen werden:
  - Laufzeit vs. Platzbedarf
  - Laufzeit vs. Portabilität (= Plattformunabhängigkeit)
  - Entwicklungszeit vs. Robustheit, Zuverlässigkeit
  - Entwicklungszeit vs. Funktionalität
  - Funktionalität vs. Benutzbarkeit
  - Kosten vs. Wiederverwendbarkeit
  - „Good, fast, cheap. Pick any two.“
- Sobald auch die nichtfunktionalen Anforderungen klar sind, kann eine erste Architektur in Angriff genommen werden.

Methodik des Entwurfs

Es gibt zwei unterschiedliche Entwurfsmethoden:

1. Modularer Entwurf (engl.: *modular design*)
2. Objekt-orientierter Entwurf (engl.: *object-oriented design*)
   - OO-Entwurf erweitert modularen Entwurf um Vererbung, Polymorphie und Datenmodellierung

Karlsruher Institut für Technologie

3.2 Modularer Entwurf (MD)

Siehe auch Balzert, Band 1, LE 32

3.2 Modularer Entwurf

- Systemarchitektur beim modularen Entwurf:
  1. **Modulführer** (Grobentwurf, *module guide*, software architecture):
     - Gliederung in Komponenten (Module) und Subsysteme
     - Beschreibung der Funktion jedes Moduls
     - Benutzt Architekturstile (und ggf. Entwurfsmuster), z.B. Schichtenarchitektur oder Fließbandarchitektur.
  2. **Modulschnittstellen** (engl. *module interfaces*):
     - Genaue Beschreibung der von jedem Modul zur Verfügung gestellten Elemente (Typen, Variablen, Unterprogramme etc.), informell oder formal.
     - Für Module mit Ein-/Ausgabe auch genaue Beschreibung der entsprechenden Formate (z.B. mit XML oder Grammatiken)

3.2 Modularer Entwurf

- Systemarchitektur beim modularen Entwurf (Forts.):
  3. **Benutztrelation** (engl. *uses relation*)
     - Gliederung in Komponenten (Module) und Subsysteme
     - Beschreibt, wie sich Module und Subsysteme untereinander benutzen. Die Benutztrelation sollte ein azyklischer, gerichteter Graph sein, damit inkrementeller Aufbau und inkrementelles Testen möglich sind.
  4. (optional) **Feinentwurf** (engl. *detailed design*)
     - Beschreibung der modul-internen Datenstrukturen und Algorithmen;
     - Bei Implementierung in Assembler auch vollständige Programmierung der Module in Pseudocode.
     - Der Pseudocode ist in einer höheren, meist hypothetischen Programmiersprache abgefasst (z.B. mit Steuerungsstrukturen entliehen von C und Anweisungen in natürlicher Sprache) und wird in der Implementierungsphase von Hand in Assembler umgesetzt.

3.2 Modularer Entwurf

- **Externer Entwurf:** Grobentwurf und Modulschnittstellen
- **Interner Entwurf:** Benutztrelation und Feinentwurf
- Die Konzepte des modularen Entwurfs behalten ihre Gültigkeit für objektorientierten Entwurf.

3.2.1 Anforderungen an das Modulkonzept

- Module sollen unabhängig voneinander bearbeitet und benutzt werden können.
  - Ein Modul sollte ohne Kenntnis der späteren Nutzung entworfen, implementiert, getestet und überarbeitet werden können;
  - Die Implementierung eines Moduls sollte möglich sein, ohne etwas über Implementierungsdetails anderer Module wissen zu müssen und ohne das Verhalten anderer Module zu beeinflussen;
  - Ein Modul sollte ohne Kenntnis seines inneren Aufbaus benutzt werden können (Kapselung der inneren Strukturen).

3.2.1 Anforderungen an das Modulkonzept

- Module sollen unabhängig voneinander bearbeitet und benutzt werden können. (Forts.)
  - Normalfall: Ein Modul enthält mehrere Unterprogramme, die eine Datenstruktur manipulieren; direkter Zugriff anderer Module auf diese Datenstruktur ist nicht möglich (Datenkapselung) (ähnlich einem Objekt).
  - Starke Kohäsion (Zusammenhalt, Bindung) innerhalb eines Moduls; Kohäsion zwischen Modulen geringer.
  - Ein Modul sollte einfach genug sein, um für sich voll verstanden werden zu können

3.2.2 Das Modul

Def. Modul (engl. *module*):
Ein Modul ist eine Menge von Programm-Elementen, die nach dem *Geheimnisprinzip* gemeinsam entworfen und geändert werden.

- Programmelemente sind Typen, Klassen, Konstanten, Variablen, Datenstrukturen, Prozeduren, Funktionen, Prozesse (Fäden), Prozess-Eingänge, Makros etc.

3.2.2 Das Modul

Def. Geheimnisprinzip, Kapselungsprinzip (*information hiding principle*):
Jedes Modul verbirgt eine wichtige Entwurfsentscheidung hinter einer wohldefinierten Schnittstelle, die sich bei einer Änderung der Entscheidung nicht mit ändert.
(David Parnas)

- Grund: Nur was verborgen und unbenutzt ist, kann ohne Risiko geändert werden.

Beispiel: Klasse für eine Strecke (1)
Ohne Kapselung oder Geheimnisprinzip

- Strecke: Anfangs- und Endpunkt, feste Länge

```
public class StreckeOhneGeheimnis {
  public int startpunkt_x;
  public int startpunkt_y;
  public int endpunkt_x;
  public int endpunkt_y;
}
```

- Benutzendes Objekt muss die Klasse „durch und durch“ kennen.

Beispiel: Klasse für eine Strecke (2)
Mit Kapselung – erster Versuch

```
public class StreckeMitKapselung1 {
  private int startpunkt_x; private int startpunkt_y;
  private int endpunkt_x; private int endpunkt_y;

  public int getStartpunkt_x() { … }
  public void setStartpunkt_x(int startpunkt_x) { … }

  public int getStartpunkt_y() { … }
  public void setStartpunkt_y(int startpunkt_y) { … }
  …
}
```

- Direkter Zugriff auf Koordinaten verhindert, Kapselung auf Attribute.
- Semantische Kapselung?

Beispiel: Klasse für eine Strecke (3)
Mit Kapselung – zweiter Versuch

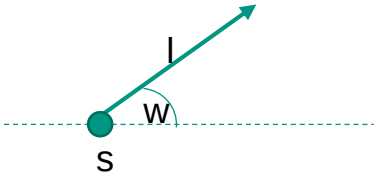
```
public class StreckeMitKapselung2 {
  private Punkt startpunkt;
  private Punkt endpunkt;

  public Punkt getStartpunkt() { return startpunkt; }
  public void setStartpunkt(Punkt startpunkt) { … }
  public double getLaenge() {
    int xdist = endpunkt.getX() - startpunkt.getX();
    int ydist = endpunkt.getY() - startpunkt.getY();
    return Math.sqrt(xdist * xdist + ydist * ydist);
  }
}
```

Klasse Punkt kapselt zwei Koordinaten X und Y.

- Semantische Kapselung: Interne Repräsentation der Strecke nach außen nicht sichtbar und Berechnung der Länge ebenso.

Beispiel: Klasse für eine Strecke (4)
Mit Kapselung – Änderung der Interna

```
public class StreckeMitKapselung3 {
  private Punkt startpunkt;
  private double laenge, winkel;

  public Punkt getEndpunkt() {
    // errechne Endpunkt aus Länge und Winkel
    … }
  public void setEndpunkt(Punkt endpunkt) {
    // errechne Länge und Winkel aus Endpunkt
    … }
  public double getLaenge() {
    return laenge;
  }
}
```



3.2.2 Das Modul

- Voraussehbare Änderungen sollten ohne Schnittstellenänderungen möglich sein. Wenn Schnittstellen sich nicht ändern, muss der Rest des Systems nicht angepasst werden.
- Weniger voraussehbare Änderungen können Schnittstellenänderungen erfordern, aber nur bei wenigen Modulen.
- Nur sehr seltene Änderungen sollten Schnittstellen von oft benutzten Modulen beeinflussen.
- Anmerkung 1: Oft führt das Geheimnisprinzip verkürzt dazu, dass Attribute privat sein sollen und nur durch Methoden geändert werden können; das ist eine notwendige, aber nicht hinreichende Bedingung.
- Anmerkung 2: Was bedeutet „deprecated“ (abgelehnt, zurückgestellt) in Java-Dokumentation?

3.2.2 Das Modul

- Vorgehensweise beim Entwurf nach dem Geheimnisprinzip:
  - Bilde Liste von Entwurfsentscheidungen,
    - die schwierig sind (d. h. solche, die man leicht falsch macht)
    - oder die sich voraussichtlich ändern werden
  - Weise jede Entscheidung einem Modul zu und definiere die Modulschnittstellen so, dass sie sich bei einer Änderung der Entscheidungen nicht mit ändern
    - Damit wird jede änderbare Entscheidung hinter einer Schnittstelle verborgen

3.2.2 Das Modul

- Kandidaten für Verbergung (Entwurfsentscheidungen, Geheimnisse der Module):
  - Der Klassiker: Datenstrukturen (Wahl der Datenstruktur, Größe) und Implementierung von Operationen an diesen Datenstrukturen
  - maschinennahe Details (z. B. Gerätetreiber, Steuerung von Ein-/Ausgabe, Zeichencodes und deren Ordnung, etc.)
  - betriebssystemnahe Details (Ein-/Ausgabeschnittstellen, Dateiformate, Netzwerkprotokolle, Kommandosprache, etc.)
  - Grundsoftware wie Datenbanken, Oberflächen-Bibliotheken, o.ä.

3.2.2 Das Modul

- Kandidaten für Verbergung: (Forts.)
  - Ein-/Ausgabeformate: nur ein Modul kennt ein geg. Format, der Rest greift programmatisch auf die Eingabe-Elemente (evtl. in interner Repräsentation) zu; analog für Ausgabe
  - Benutzungsschnittstellen: Kommandoschnittstelle, graphische Oberfläche, Gesten-gesteuerte Oberflächen, Web, Sprachsteuerung; Kombinationen davon
  - Text von Dialogen, Beschriftungen, Ausgaben und Fehlermeldungen (Sprache austauschbar machen!)
  - Größe von Datenstrukturen
  - Reihenfolge der Verarbeitung

3.2.3 Modulführer

- Entwurf und Dokumentation der Modulstruktur:
  - Ergebnis: Modulführer (oder Grobentwurf)
  - Der Modulführer beschreibt für jedes Modul oder Subsystem:
    - die Entwurfsentscheidungen, die das Geheimnis des Moduls/Subsystems sind
    - die Funktion des Moduls/Subsystems
    - die Gliederung von Subsystemen in Module und andere Subsysteme/Pakete (Zerlegungsstruktur, Bestandshierarchie)

3.2.3 Modulführer

- Ein sorgfältig aufgestellter Modulführer hat folgende Vorteile
  - Vermeidet Duplikation
  - Vermeidet Lücken
  - Liefert eine Zerlegung des Problems
  - Erleichtert das Auffinden von betroffenen Modulen während der Wartung

3.2.4 Modulschnittstellen

- Entwurf und Dokumentation der Modulschnittstellen:
  - Ergebnis: „Black Box“-Beschreibung jedes Moduls
    - Genau die Information, die sowohl für die Benutzung als auch für die Implementierung des Moduls erforderlich ist und nicht mehr.
    - Die Schnittstelle ist invariant bezüglich voraussehbarer Änderungen.

3.2.4 Modulschnittstellen

- Beschreibung der Modulschnittstellen besteht aus:
  - Liste der öffentlichen Programmelemente
  - Ein-/Ausgabeformate (falls E/A-Modul)
  - Parameter und Rückgabetypen der Unterprogramme/Operationen
  - Beschreibung des Effekts der Unterprogramme
  - Zeitverhalten, Genauigkeit, Speicherplatzbedarf und andere Qualitätsmerkmale, wo erforderlich
  - Fehlerbeschreibung und Fehlerbehandlung
  - Ausnahmen, die ausgelöst und nicht behandelt werden.

3.2.5 Fallstudie: KWIC-Index

- Aufgabe: Ein Programm zur Erzeugung eines KWIC-Indexes (permutierter Index; KWIC=Keyword in context)
  - Eingabe:
    - Folge von Zeilen („Titeln“)
    - Jede Zeile ist eine Folge von Wörtern
    - Jedes Wort ist eine Folge von Zeichen
  - Ausgabe:
    - Aus jeder Zeile werden alle zirkulären Verschiebungen erzeugt (durch wiederholtes Entfernen des ersten Wortes und Anhängen am Ende der Zeile).
    - Die zirkulären Verschiebungen werden alphabetisch sortiert ausgegeben.

3.2.5 Fallstudie: KWIC-Index

- Beispiel:
  - Gegeben seien folgende Titel:
    - **Harry Potter und der Orden des Phönix**
    - **Der Flug des Phönix**
    - **Der Orden der Schwerter**
    - **Orden und Klöster**
  - Bei Weglassen von Verschiebungen, die mit der, die, das, des, und, … beginnen, entsteht der folgende KWIC-Index:

3.2.5 Fallstudie: KWIC-Index

Der	FLUG des Phönix
	HARRY Potter und der Orden des Phönix
Orden und	KLÖSTER
	ORDEN und Klöster
Harry Potter und der	ORDEN des Phönix
Der	ORDEN der Schwerter
Der Flug des	PHÖNIX
Harry Potter und der Orden des	PHÖNIX
Harry	POTTER und der Orden des Phönix
Der Orden der	SCHWERTER

Ein KWIC Index (auch permutierter Index) eignet sich für manuelle und maschinelle Suche in einer großen Anzahl von Titeln. („Google für Arme“)

3.2.5 Fallstudie: KWIC-Index

Weiteres Beispiel:

Eingabe: **„abs: integer absolute value“**

zirkuläre Verschiebungen:

abs: integer absolute value

absolute value, abs: integer

integer absolute value, abs

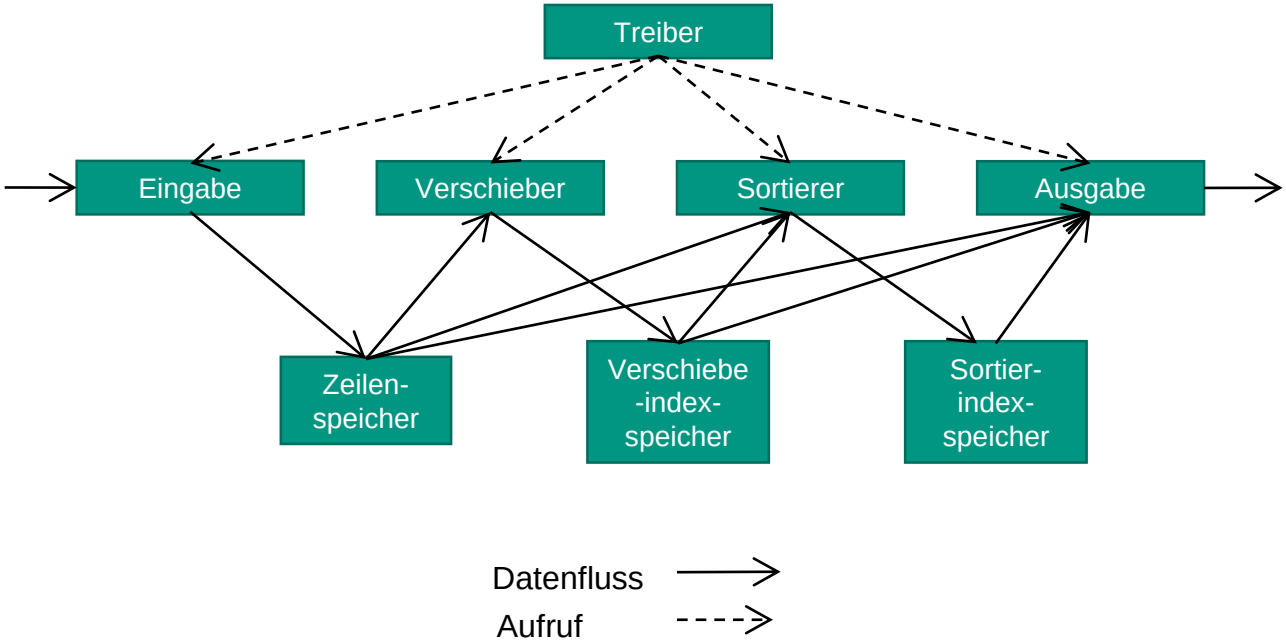
value, abs: integer absolute

oder sauber formatiert:

	abs: integer absolute value
abs: integer	absolute value
abs:	integer absolute value
abs: integer absolute	value

3.2.5 Fallstudie: KWIC-Index

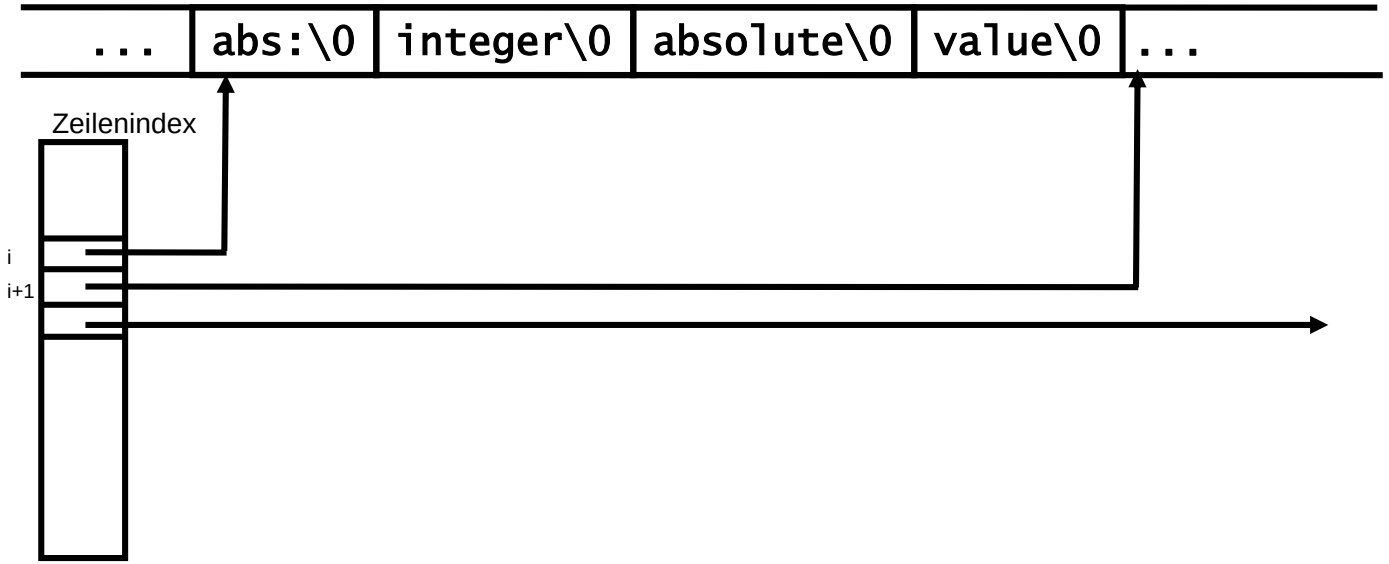
- Entwurf 1 (Datenflussentwurf):



3.2.5 Fallstudie: KWIC-Index

- Eingabe:
  - Programm liest die Zeilen von der Eingabe und speichert sie im Hauptspeicher.
  - Zeichen werden vier pro Speicherwort abgelegt, ein spezielles Zeichen markiert das Ende jedes Wortes.
  - Ein Index wird erzeugt, der den Anfang jeder Zeile angibt.

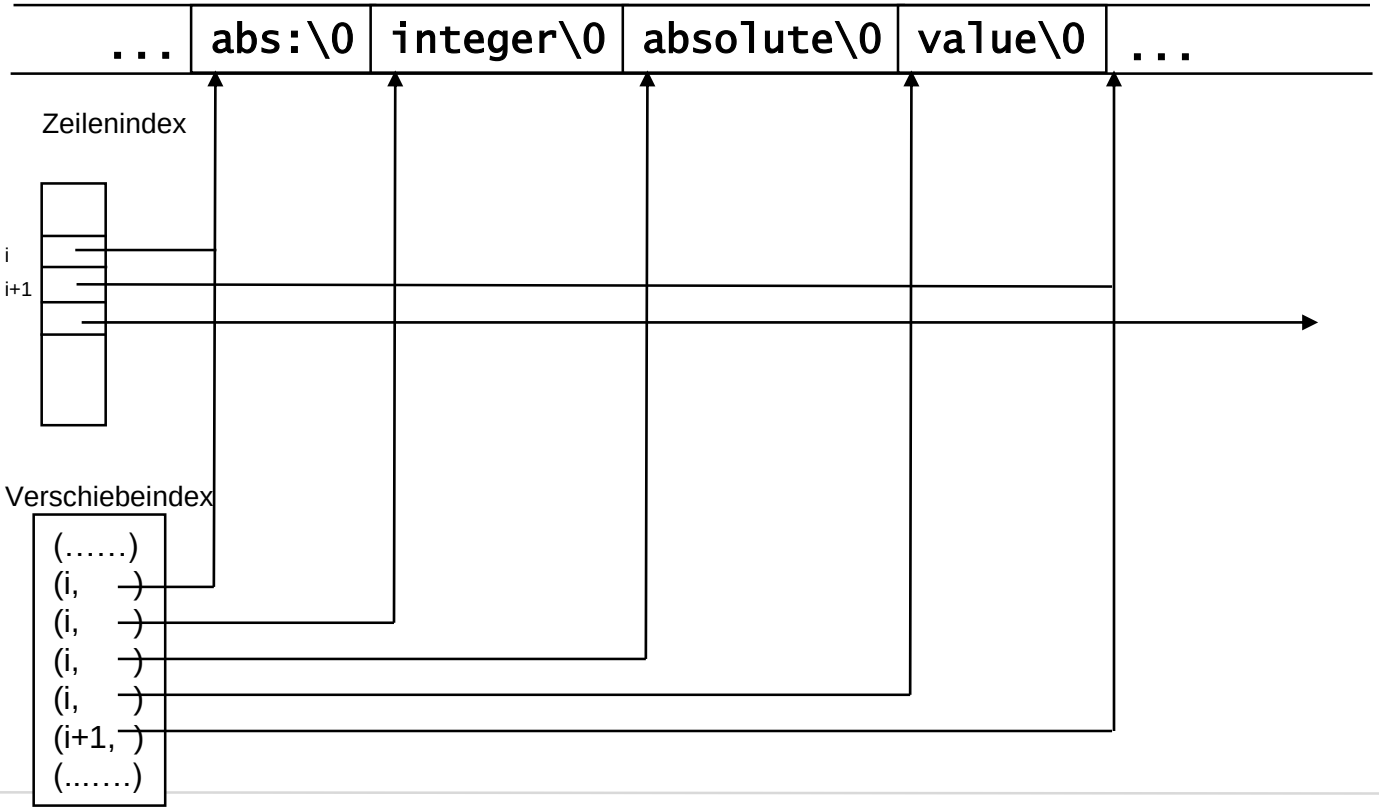
3.2.5 Fallstudie: KWIC-Index



3.2.5 Fallstudie: KWIC-Index

- Verschieber:
  - Wird aufgerufen, nachdem Eingabe vollständig ist.
  - Erzeugt einen Index, der alle zirkulären Verschiebungen enthält.
  - Jede Verschiebung besteht aus (`<Zeilennr>`, `<Adresse>`)
    - `<Zeilennr>` ist Index der Zeile
    - `<Adresse>` ist Adresse des Anfangszeichens der zirkulären Verschiebung
  - Diese Lösung vermeidet eine Vervielfachung der Titel.

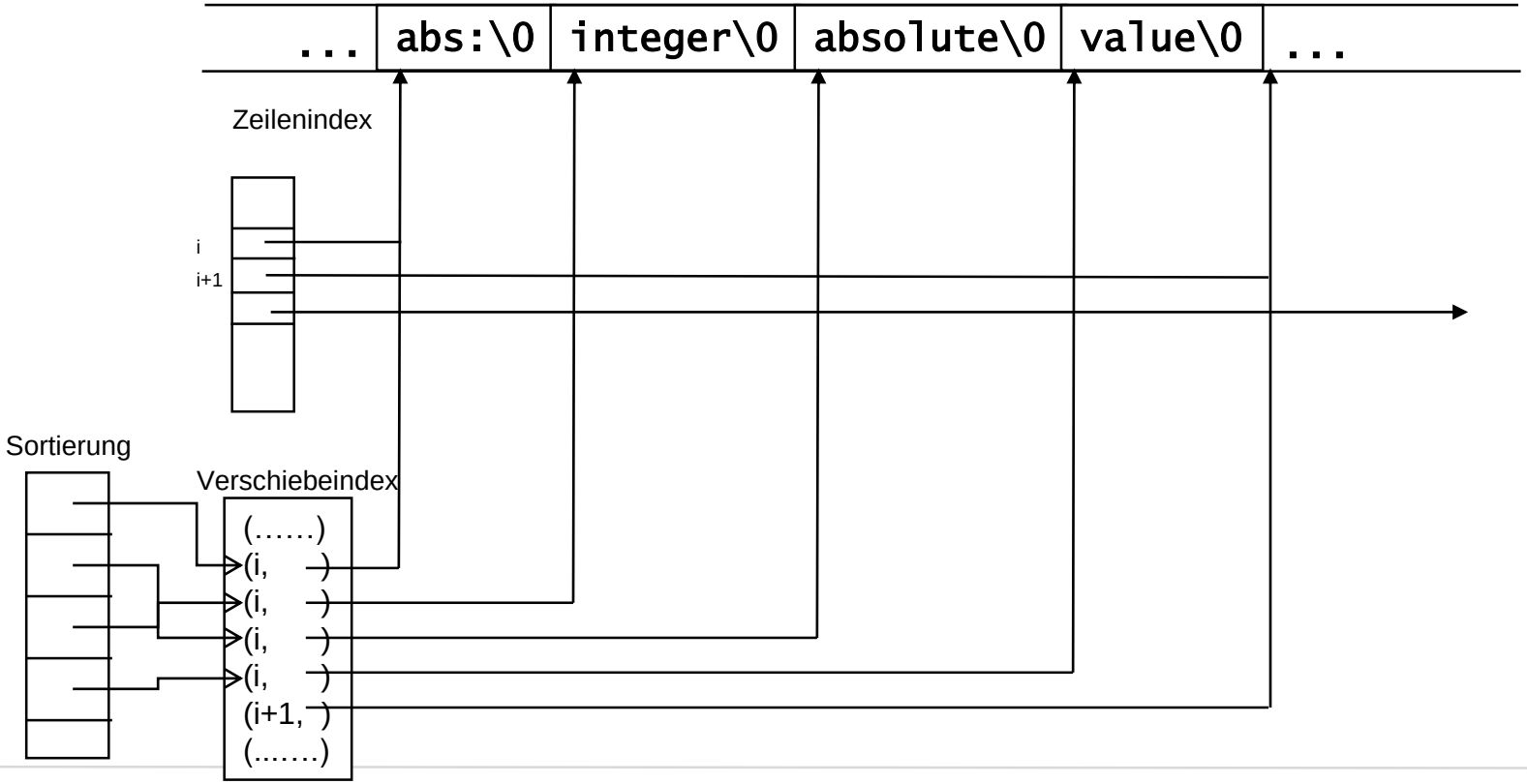
3.2.5 Fallstudie: KWIC-Index



3.2.5 Fallstudie: KWIC-Index

- Sortierer:
  - Modul operiert an Datenstrukturen, die von den ersten beiden Modulen erzeugt wurden.
  - Bildet einen Verschiebungsindex wie das Verschiebungsmodul, außer dass die Verschiebungen alphabetisch sortiert sind
  - oder bildet ein 1-d-Feld, in dem die Nummern der Verschiebungen in sortierter Reihenfolge stehen (Permutation des Verschiebungsindexes).
  - Auch damit wird eine Vervielfachung der Daten vermieden.

3.2.5 Fallstudie: KWIC-Index



3.2.5 Fallstudie: KWIC-Index

- Ausgabe:
  - Modul operiert an den von den Modulen Eingabe, Verschieber und Sortierer erzeugten Datenstrukturen.
  - Druckt eine formatierte Ausgabe aller zirkulären Verschiebungen.

3.2.5 Fallstudie: KWIC-Index

- Treiber:
  - Ruft Untermodule auf.
  - Behandelt Fehlermeldungen.

- Die genaue Spezifikation der Schnittstellen fehlt noch!

3.2.5 Fallstudie: KWIC-Index

- Dieser Entwurf würde funktionieren.
- Welche Probleme gibt es mit diesem Entwurf?

3.2.5 Fallstudie: KWIC-Index

- Wichtige Überlegungen (Teil 1):
  - Die Speicherung der Titel kann sich ändern
    - Hauptspeicher
    - Platte
    - gemischt
    - andere Speicherwortgröße
  - Die Speicherstruktur kann sich ändern
    - gepackte/ungepackte Darstellung
    - Zeichensätze (ASCII, Unicode)
    - Speicherung als Liste von Wörtern auf der Halde
    - Vermeidung von Duplikaten
    - Ignorieren von unwichtigen Wörtern wie a, an, the, that, of etc.

3.2.5 Fallstudie: KWIC-Index

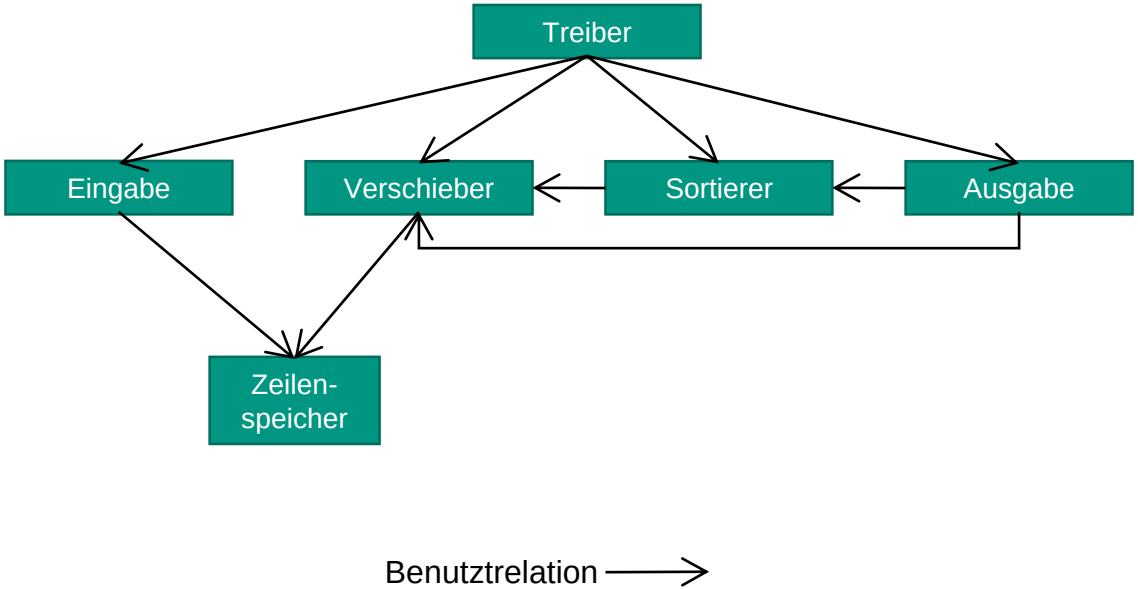
- Wichtige Überlegungen (Teil 2):
  - Die Entscheidung, einen Index für die Verschiebungen anstatt die Verschiebungen explizit zu erzeugen, kann sich ändern.
  - Die Entscheidung über die Sortierung kann sich ändern
    - Einträge nach Bedarf suchen
    - partiell sortieren (etwa nach den ersten Buchstaben).
  - Die Ein-/Ausgabeformate können sich ändern (ASCII, PS, PDF, HTML, XML, ...)

3.2.5 Fallstudie: KWIC-Index

- Im Entwurf 1 müssen für jede Änderung außer der letzten alle Module überarbeitet werden!

3.2.5 Fallstudie: KWIC-Index

- Entwurf 2:



3.2.5 Fallstudie: KWIC-Index

- Zeilenspeicher: Speicherung für Zeilen
  - Die Idee hier: anstelle eine Datenstruktur offen zu legen, wird eine Zugriffsschnittstelle angeboten. Hier besonders einfach, da Titel, Wörter und Zeichen durchgezählt werden können.
  - Prozedur `setzeZeichen(r,w,c,d)` setzt das c-te Zeichen des w-ten Wortes der r-ten Zeile zu d.
  - Funktion `holeZeichen(r,w,c)` gibt das c-te Zeichen des w-ten Wortes der r-ten Zeile zurück
  - Funktion `holeZeilenZahl()` gibt die Anzahl der Zeilen zurück

3.2.5 Fallstudie: KWIC-Index

- Zeilenspeicher:
  - Funktion `holeWörterZahl(r)` gibt die Anzahl der Wörter in der Zeile r zurück
  - Funktion `holeZeichenZahl(r,w)` gibt die Anzahl der Zeichen im w-ten Wort der r-ten Zeile zurück.
- Die interne Darstellung der Zeilen ist völlig beliebig.
  - Die Schnittstelle für das Auslesen der Titel bietet sog. Iteratoren (man kann über Zeichen von Wörtern, Wörtern in Zeilen, und Zeilen iterieren)
  - Übungsaufgabe: Schreibe ein Programm, das die Zeilen in umgekehrter Speicherreihenfolge ausgibt.
  - Übungsaufgabe: Entwerfe die Datenstrukturen, um diese Schnittstelle zu implementieren.
  - Übungsaufgabe: Mit welcher Datenstruktur kann das Speichern von Duplikaten vermieden werden? Wie findet man schnell, ob ein geg. Wort schon gespeichert ist?
  - Übungsaufgabe: Schreibe ein Programm, das Titel einliest und im Zeilenspeicher ablegt.

3.2.5 Fallstudie: KWIC-Index

Ausgabe der Zeilen mit 3 Iteratoren:

```
for (r = 0, r < holeZeilenZahl(), r++) {
    for (w = 0, w < holeWörterZahl(r), w++) {
        for (c = 0, c < holeZeichenZahl(r,w), c++)
            printc(holeZeichen(r,w,c));
        printc(' ,);
     }
    printc('\n‘);
}
```

3.2.5 Fallstudie: KWIC-Index

- Verschieber:
  - Prozedur `VSInit()` initialisiert den Verschieber.
  - Funktion `VSVerschiebungsZahl()` gibt die Anzahl der Verschiebungen an.
  - Funktion `VSholeZeichen(l,w,c)` gibt das c-te Zeichen des w-ten Wortes der l-ten **Verschiebung** zurück.

3.2.5 Fallstudie: KWIC-Index

- Verschieber fortgesetzt:
  - Funktion `VsholeWörterZahl(l)` gibt die Anzahl der Wörter in der l-ten Verschiebung an.
  - Funktion `VsholeZeichenZahl(l,w)` gibt die Anzahl der Zeichen im w-ten Wort der l-ten Verschiebung zurück.
  - Analog zu Zeilenspeicher, aber jetzt für Verschiebungen.

3.2.5 Fallstudie: KWIC-Index

- Verschieber:
  - kann einen Index erzeugen
  - die Verschiebungen direkt erzeugen
  - oder Verschiebungen nach Bedarf berechnen.
  - Übungsaufgabe: Verschiebungen sollten für Stoppwörter (der, die, das, ein, einer, eines, von, und, usw.; engl. „stop words“) nicht erzeugt werden. Wie würden Sie dieses Problem lösen?
    Wichtige Randbedingung: Die ausgegebenen Verschiebungen sollten die unwichtigen Worte weiterhin enthalten, um lesbar zu bleiben.

3.2.5 Fallstudie: KWIC-Index

- Sortierer:
  - Prozedur `alphabetisiere()` initialisiert den Sortierer.
  - Funktion `holeIte(i)` liefert die Nummer der Verschiebung, die an der i-ten Stelle in der Sortierung steht.
- Der Sortierer kann in `alphabetisiere()` vollständig sortieren oder auch nur vorsortieren (z.B. nach erstem Buchstaben)
- Die Sortierung kann auch nach und nach berechnet werden (z.B. jedes Mal, wenn ein neuer Titel hinzugfügt wird)

3.2.5 Fallstudie: KWIC-Index

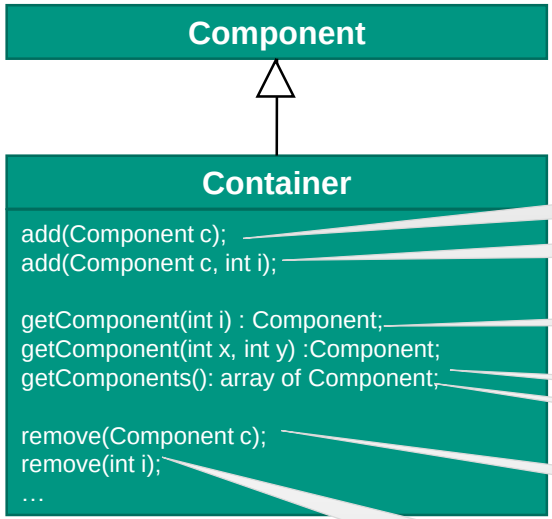
- Eingabe:
  - Prozedur einlesen() liest die Zeilen von der Eingabe und schreibt zeichenweise in den Zeilenspeicher. Eingabeformat wie anfangs besprochen.

- Ausgabe
  - Prozedur verschiebungenAusgeben() druckt alle Verschiebungen auf dem Bildschirm aus.

3.2.5 Noch ein Beispiel: java.awt.Component

- Die Klasse `java.awt.Component` erlaubt das Hinzufügen, Entfernen und Auslesen von Komponenten über eine Zugriffsschnittstelle.
- Dadurch kann die Speicherung der Komponenten in der Klasse `Component` geändert werden, ohne dass andere Anwendungen davon betroffen sind
  (z.B. als Feld, Liste, geordnete Menge).
- Durch die Schnittstellen wird eine einfache Verwendung von unterschiedlichen Anordnern ermöglicht. Viele Anordner brauchen die Komponenten aber in einer linearen Ordnung (z.B. FlowLayout).
- Auch hier ist es leicht, über die Komponenten zu iterieren.

Beispiel: java.awt.Component



Übung: Schreibe einen Iterator, der alle Komponenten in der gespeicherten Reihenfolge ausgibt, ohne die dritte getComponents()-Methode zu benutzen!

Füge Komponente c an das Ende des Containers an (lineare Ordnung)

Füge Komponente c an Position i des Containers an (verschiebe Rest)

Hole Komponente an Position i

Hole Komponente an Maus-Position x,y

Hole alle Komponenten des Containers

Entferne Komponente c aus Container

Entferne Komponente an Position i des Containers

3.2.6 Das Modul in Programmiersprachen

- Programmiersprachen wie C, C++, Ada und Modula bieten sprachliche Mittel, um Module zu beschreiben.
- Generelle Idee: Zerlege Modul in
  - Schnittstelle und
  - Implementierung gleichen Namens.
- Die Schnittstelle enthält die öffentlichen, nach außen sichtbaren Elemente, wie Konstanten, Typdeklarationen, Variablen, und Köpfe von Unterprogrammen (Signaturen).
- Die Implementierung enthält private Elemente, sowie die Implementierungen der öffentlichen und privaten Unterprogramme.
- Die Schnittstelle von Modulen ist also den Schnittstellen in OO-Sprachen ähnlich.

3.2.6 Das Modul in Programmiersprachen

- Und wie geht das in C/C++?
  - C/C++ bieten Dateiinklusion, d.h. Dateien können in andere Dateien beim Übersetzungsvorgang eingesetzt werden.
  - Die Schnittstelle eines Moduls ist in einer „.h“-Datei spezifiziert, z.B. `Verschieber.h`, `Zeilenspeicher.h`
  - Die Implementierungseinheit ist eine „.c“-Datei, die am Anfang eine Inklusionsdirektive enthält. `Verschieber.c` enthält

```
#include "Zeilenspeicher.h"
#include "Verschieber.h"
```

  - Keine Überprüfung der Namenspaarung – lediglich eine Konvention. Die Inklusionsdirektiven werden einfach durch den Inhalt der genannten Dateien ersetzt.
  - Damit hat der Übersetzer genug Information, um Aufrufe an den Zeilenspeicher zu überprüfen, und die Implementierung auf Übereinstimmung mit der Schnittstelle von Verschieber zu prüfen.

3.2.6 Das Modul in Programmiersprachen

Datei `Zeilenspeicher.h` enthält:

```
/* setzt das c-te Zeichen des w-ten Wortes
   der r-ten Zeile zu d */
void setzeZeichen (unsigned int r, unsigned int w,
                   unsigned int c, unsigned int d);

/* gibt das c-te Zeichen des w-ten Wortes
   der r-ten Zeile zurück */
unsigned int holeZeichen(unsigned int r,
                   unsigned int w, unsigned int c);

/* gibt die Anzahl der Zeilen zurück */
unsigned int holeZeilenZahl();

/* usw. */
```

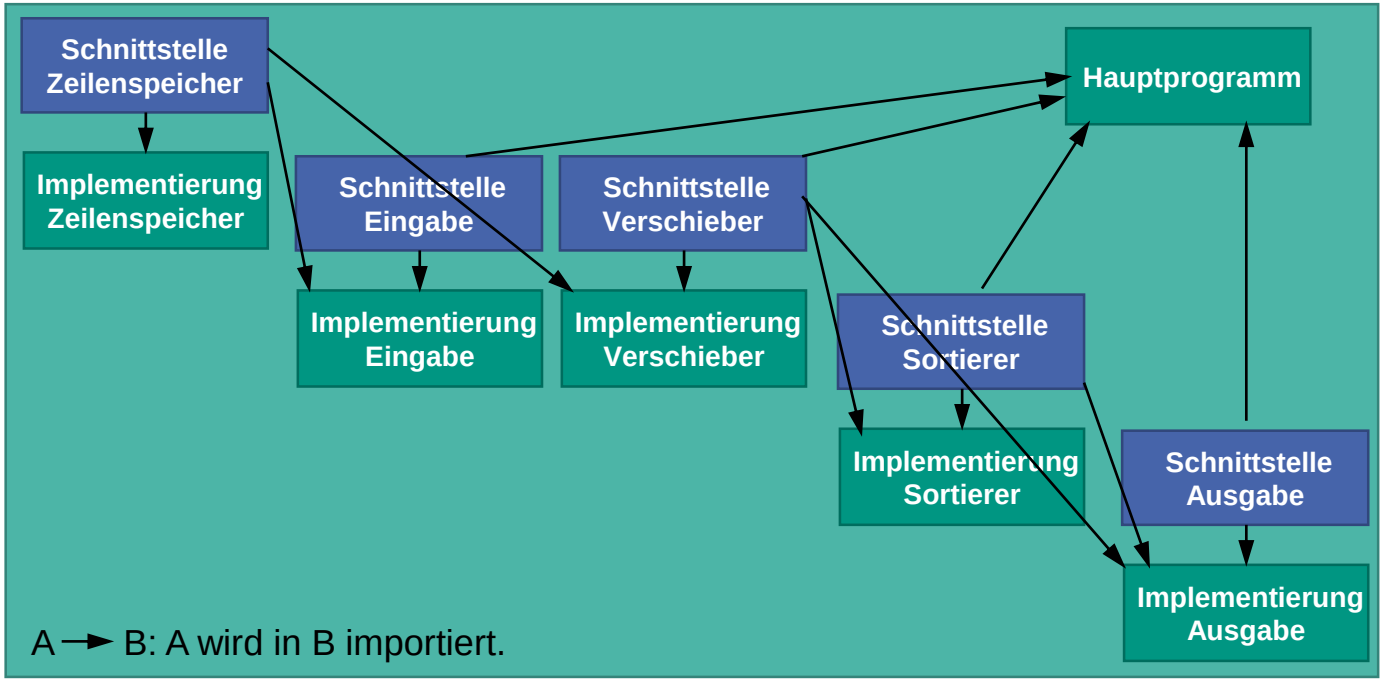
3.2.6 Das Modul in Programmiersprachen

Datei `Verschieber.h` enthält:

```
/* VSInit() initialisiert den Verschieber */
void Vsinit();

/* VSVerschiebungszahl gibt die Anzahl der
   Verschiebungen zurück */
unsigned int VSVerschiebungszahl();

/* VSholeZeichen gibt das c-te Zeichen des w-ten
   Wortes der l-ten Verschiebung zurück */
unsigned int VSholeZeichen(unsigned int l,
                  unsigned int w, unsigned int c);

/* usw. */
```

3.2.6 Das Modul in Programmiersprachen

Importierungsgeflecht für KWIC:



3.2.6 Das Modul in Programmiersprachen

- Es ist nicht möglich, ein Modul mehrfach anzulegen.
  - Das Paar von Schnittstelle und Implementierung bildet das einzige Exemplar des Moduls und seiner Datenstrukturen.
    - Wenn mehrere Exemplare nötig werden, muss das Modul kopiert und die Kopie umbenannt werden, oder aber eine Mehrfachanlegung der Datenstrukturen gleich im Modul einprogrammiert sein.
    - Dieses Problem ist bei objektorientierten Sprachen durch Mehrfachinstanziierung einer Klasse oder durch Mehrfachimplementierung einer Schnittstelle gelöst.

3.2.6 Das Modul in Programmiersprachen

- Vergleiche dazu den objektorientierten Ansatz
  - Schnittstelle kann durch mehrere Klassen implementiert sein
  - Von diesen Klassen können mehrere Objekte erzeugt werden
  - Umgekehrt können Klassen mit ausschl. statischen Attributen Einzelobjekte nachbilden; s.a. Entwurfsmuster Einzelstück
- Bei OO-Sprachen ist es auch möglich, die gleichen Bezeichner in verschiedenen Klassen zu verwenden.
  - In modularen Sprachen ist das nicht möglich – vgl. die Bezeichner `HoleZeichen` und `VSHoleZeichen`.

3.2.7 Gestaltung der Benutztrelationen

- Hierarchie-Relationen:
  - delegiert-an (delegates-to)
  - ist-ein (is-a)
  - hat-ein (has-a)
  - enthält-ein (contains, is-composed-of)
  - greift-zu-auf (accesses)
  - ist-privilegiert-zu (is-privileged-to, owns)
  - ruft auf (calls)
  - aufgerufen-von (called-by)
  - benutzt (uses)
  - stellt-Betriebsmittel-bereit

3.2.7 Gestaltung der Benutztrelationen

Def. Benutztrelation, (benutzt): Programmkomponente A benutzt Programmkomponente B genau dann, wenn A für den korrekten Ablauf die *Verfügbarkeit* einer korrekten Implementierung von B erfordert.

3.2.7 Gestaltung der Benutztrelationen

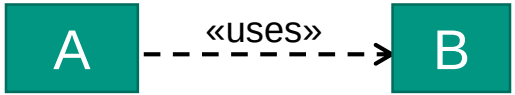
- Verfügbarkeit kann heißen:
  - A delegiert Arbeit nach B (Delegationsrelation)
  - A greift auf Variable von B zu
  - A ruft B auf, wobei A korrekten Ablauf von B erfordert. (Ein Aufruf impliziert nicht notwendigerweise eine Benutzung. Z. B. ruft ein Betriebssystem ein Anwenderprogramm auf, aber es benutzt es nicht.)
  - A legt eine Instanz eines Typs aus B an
  - A steuert B durch Unterbrechungen oder Ereignisse an (und erfordert korrekten Ablauf von B)

3.2.7 Gestaltung der Benutztrelationen

- Benutztrelation
  - Die Benutztrelation kann eine Halbordnung oder eine Totalordnung sein.
  - Paradebeispiel für solche Benutztrelationen sind schichtenorientierte Systeme, z. B.
    - THE-Betriebssystem/Dijkstra
    - Multics-Betriebssystem (Vorläufer von Unix)
    - ISO/OSI-Protokollsuite
    - TCP/IP-Protokollsuite
    - 3-Schichten-Architektur
      <Datenhaltung, Applikationskern, Benutzungsschnittstelle>

Kriterien für die Konstruktion der Benutzthierarchie

- A soll B benutzen, wenn alle der folgenden Kriterien zutreffen:
  - A wird durch die Benutzung von B einfacher.
  - B wird nicht wesentlich komplexer dadurch, dass es A nicht benutzen darf.
  - Es gibt *mind. eine* nützliche Untermenge, die B, aber nicht A enthält.
  - Es gibt *keine* Untermenge, die A, aber nicht B enthält.

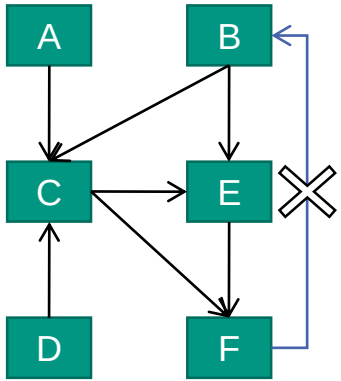


3.2.7 Gestaltung der Benutztrelationen

Def.: Wenn die Benutztrelation zyklenfrei ist, heißt sie **Benutzthierarchie**.

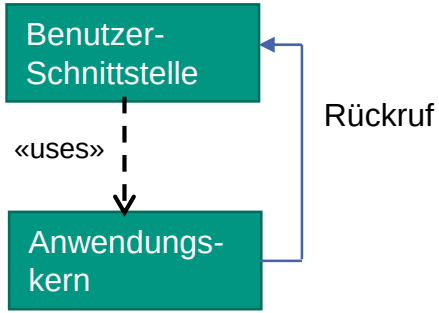
3.2.7 Gestaltung der Benutztrelationen

- Benutztrelation zwischen Modulen sollte hierarchisch, d. h. frei von Zyklen sein.
- Nachteil von Zyklen:
  - „Nothing works until everything works“
  - Vorteil von Zyklenfreiheit:
    - Schrittweiser Aufbau und Testen möglich
    - Bildung von Untermengen
      - Zum Einbau in andere Systeme
      - Zur eigenständigen Verwendung
      - Zur Teillieferung bei Entwicklungsverzögerungen



3.2.7 Gestaltung der Benutztrelationen

- Und was ist mit Rückrufen (callbacks)?
- Beispiel: Benutzerschnittstelle benutzt den Anwendungskern; aber der Kern muss bei bestimmten Ereignissen auch Aufrufe in die Benutzerschnittstelle durchführen dürfen, z.B. um eine Anzeige zu aktualisieren.



3.2.7 Gestaltung der Benutztrelationen

- In der Regel wird dies so organisiert, dass die Benutzerschnittstelle die Adressen von Unterprogrammen, die bei den Ereignissen aufzurufen sind, dem Kern bei der Initialisierung in einem Aufruf übergibt
- Das Gleiche kann durch Übergabe von Objekten oder Klassen geschehen, die einen Satz von Methoden zum Aufruf zur Verfügung stellen (z.B. `registerMouseListener`)
- Es handelt sich hier aber um *keine* Nutzung der Benutzerschnittstelle durch den Kern; der Kern benötigt zum korrekten Ablauf keine Benutzerschnittstelle (ähnlich dem Betriebssystem, das Anwendungen aufruft, aber sie nicht benötigt.)
- D.h. bei einem Rückruf ist die Benutztrelation nicht zyklisch

3.2.8 Bibliographie zum Entwurf

- Zur Historie
  - Prof. Dr. David Lorge Parnas
    *10.2.1941 in New York
    Communications Research
    Laboratory
    University Hamilton, Ontario, Canada
  - Erfinder des Geheimnisprinzips
    (information-hiding) (1971)
  - Wegbereiter des Modularen Entwurfs (1972)
    - Spezifikation von Moduln
    - Trennung von Spezifikation und Implementierung
    - modulare Software-Architektur
  - Beiträge zur Disziplin Software-Technik (seit 1975)
  - Beiträge zur Qualitätssicherung von Software

3.2.8 Bibliographie zum Entwurf

- Pflichtlektüre:
  - „Designing Software for Ease of Extension and Contraction“, David L. Parnas, IEEE Trans. On Software Engineering, SE-5(2), März 1979.
- „On the Criteria to be Used in Decomposing Systems into Modules“, D. L. Parnas, Communications of the ACM, 15(12), Dezember 1972
- „The Modular Structure of Complex Systems“, Davis L. Parnas, IEEE Trans. on Software Engineering, SE-11(3), März 1985
- „A Rational Design Process: How and Why to Fake it“, D. L. Parnas, P. C. Clements, IEEE Trans. on Software Engineering SE-12(2), Februar 1986

Karlsruher Institut für Technologie

3.3 Objektorientierter Entwurf (OOD)

3.3 Objektorientierter Entwurf

- Im objektorientierten Entwurf behalten die Prinzipien des modularen Entwurfs ihre Gültigkeit
  - Flexibilisierung der Software durch das Geheimnisprinzip
- Schnittstellen verbergen Entwurfsentscheidungen, die veränderbar bleiben sollen

3.3 Objektorientierter Entwurf

- Externer Entwurf
  - Die Analoga zum Modul sind die Klasse und das Paket
  - Im Paket werden mehrere Klassen, die gemeinsame Entwurfsentscheidungen kapseln, zusammengefasst.
  - Eine einzelne Klasse kann auch ein ganzes Modul verwirklichen; i.d.R. braucht man aber mehrere Klassen pro Modul.

3.3 Objektorientierter Entwurf

- Externer Entwurf
  - Anstelle des Modulführers tritt der Paket- und Klassenführer, i.d.R. als UML-Klassen- und UML-Paketdiagramm mit erläuterndem Text
    - dokumentiert die Entwurfsentscheidungen
  - Das Analogon zu den Modulschnittstellen sind die Schnittstellen der Klassen, abstrakte Klassen und reine Schnittstellen
    - *interfaces*

3.3 Objektorientierter Entwurf

- Interner Entwurf
  - Die Benutztrelation wird auf der Ebene von Paketen und allein stehenden Klassen (die nicht selbst in Paketen liegen) dokumentiert.
  - Der Feinentwurf liefert die Beschreibung der modulinternen Datenstrukturen und Algorithmen, sowie Pseudocode wo erforderlich, wie beim modularen Entwurf.

3.3 Objektorientierter Entwurf

- Allerdings ergeben sich im OO-Entwurf zusätzliche Möglichkeiten, die im modularen Entwurf schwieriger zu verwirklichen sind.
  - Mehrfach-Instanzierung von Klassen,
  - Vererbung und Polymorphie,
  - Variantenbildung in einem Programm durch Mehrfachimplementierung einer Schnittstelle

3.3 Objektorientierter Entwurf

- Hierzu haben sich neue Strukturen und Architekturen entwickelt, die **Entwurfsmuster** genannt werden.
- Diese werden in den folgenden Vorlesungen behandelt.
- Es gibt sowohl objektorientierte als auch klassische Entwurfsmuster, die ohne OO auskommen.
- Entwurfsmuster und Architekturstile kommen als nächstes!

3.3 Objektorientierter Entwurf

- Zur Historie
  - Grady Booch
    *27.2.1955 in Texas
    Chief Scientist
    Rational Software Corporation
  - Pionier auf dem Gebiet des
    modularen und objektorientierten Softwareentwurfs
    - 1983: Buch Software Engineering with Ada
    - 1987: Buch Software Components with Ada
    - 1991/94: Buch Object Oriented Design with Applications
  - Pionier wiederverwendbarer Bibliotheken
    - 1987: Komponentenbibliothek in Ada
    - 1991: Klassenbibliothek in C++.