

Konzepte der Programmiersprachen  
Strukturierungsmechanismen für größere  
Programmsysteme  
Lehrstuhl Prof. Plödereder

Eduard Wiebe

Institut für Softwaretechnologie  
Abteilung Programmiersprachen und Übersetzerbau

Sommersemester 2007

# Strukturierungsmechanismen für größere Programmsysteme

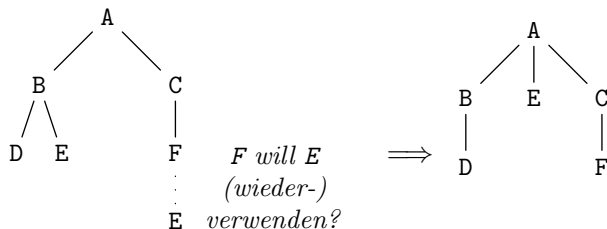
## Anforderungen:

- ▶ Aufteilbarkeit in möglichst separat entwickelbare und testbare Teile
- ▶ Beschreibbarkeit der Abhängigkeit zwischen den Teilen
- ▶ Kontrollierbarkeit gegenüber „ungeplanten“ Abhängigkeiten
- ▶ Ausnutzung der Mehrfachverwendung (und Wiederverwendung) von Teilen
- ▶ Reflexion der Architektur des Systementwurfs

Welche Sprachmittel unterstützen diese Anforderungen?

# Top-Down-Entwurf (stepwise refinement)

Sprachmittel: z. B. hierarchische Schachtelung von Unterprogrammen (Pascal) oder Modulen



— Programmverschachtelung

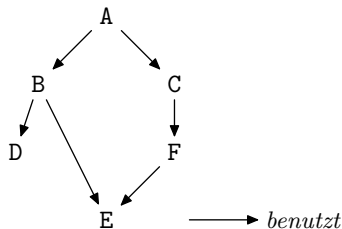
# Top-Down-Entwurf (Fortsetzung)

## Probleme:

- ▶ Gemeinsame Dienste sind von „verfeinernden“ Diensten nicht unterschieden
- ▶ Gemeinsame Dienste müssen auf einer Hierarchie-Ebene angesiedelt sein, die den Verwendern gemeinsam sichtbar ist
  - ⇒ unnötig breite Sichtbarkeit und damit Möglichkeit ungeplanter Abhängigkeit
  - ⇒ Systemarchitektur nicht mehr offensichtlich

# Bottom-Up-Entwurf u. ä.

Sprachmittel: Sammlung separater Bausteine



mit entsprechenden Abhängigkeitsbeziehungen

Baustein: Modul, Klasse, abstrakter Typ etc.

- ▶ Möglichkeit für wiederverwendbare Bausteine
- ▶ aber: globale Sichtbarkeit der Bausteine
- ▶ jedoch: Abhängigkeiten evtl. differenziert spezifizierbar

# Beschreibung der Abhängigkeiten

1. Was wird durch einen Baustein angeboten („Exporte“)
2. Was wird von einem Baustein benötigt („Importe“)  
⇒ Prüfbarkeit der Existenz und Abdeckung der Abhängigkeiten

## Exporte:

- ▶ Implizit aus der Gruppierung (Trennung von Schnittstelle und Implementierung)
- ▶ Explizite Export-Angaben  
evtl.: nach „Importeuren“ differenzierte Exporte

## Importe:

- ▶ durch Einzelangabe pro importierter Deklaration
- ▶ durch Angabe einer Gruppierung für alle jeweiligen Exporte
- ▶ durch Objekttypen / Klassen impliziert

# Unabhängige Übersetzung

Bausteine werden in beliebiger Reihenfolge übersetzt.

Linker fügt Bausteine zusammen (Abgleich von Exporten und Importen).

- ⇒ Verlust von Konsistenzprüfungen (z. B. Typ-Prüfungen, Signatur-Prüfungen) an den Grenzen der Übersetzung
- ⇒ „Notwendigkeit“ separater Werkzeuge zum Nachholen dieser Prüfungen (z. B. „lint“ für C)

# Abhängige Übersetzung

Ein Modul kann erst übersetzt werden, wenn alle Import-liefernden Module übersetzt sind.

## **Vorteile:**

Volle Konsistenzüberprüfung bei Übersetzung möglich.

## **Probleme:**

1. Schafft Projektabhängigkeiten
2. Gegenseitig abhängige Module  
⇒ Teillösung: Trennung von Spezifikation und Implementierung der Module
3. Konsequenzen bei Veränderungen von Modulen

# Konsequenzen bei Veränderung exportierender Module

... existieren natürlich bei beiden Übersetzungsmodellen

⇒ „Make“-Funktion der Sprachumgebung erforderlich

oder

⇒ Konzept der Programmbibliothek in der Sprache verankert  
(LISP, Ada, Java)

# Make

Aufgrund einer (separaten) Beschreibung der Modulabhängigkeiten wird von einem Werkzeug

- ▶ die Übersetzungsreihenfolge
- ▶ die Notwendigkeit von Neuübersetzungen

erkannt und veranlasst.

## **Pro und Contra:**

- ▶ Konsistenzprüfung über PS hinaus möglich
- ▶ Konsistenz nur bei Anwendung des Werkzeugs garantiert
- ▶ Gefahr unvollständiger/falscher Beschreibung der Modulabhängigkeiten

Sprachregeln schreiben vor, dass nach Übersetzung eines Moduls alle nicht neu übersetzten, abhängigen Module nicht importiert (oder gebunden) werden können.

- ▶ Erzwingt Anwendung der „Make“-Funktionalität
- ▶ Abhängigkeitsbeziehungen direkt in Sprachkonstrukten verankert, daher immer korrekt
- ▶ Konsistenz immer garantiert
- ▶ Automatische „Closure“ für Binder