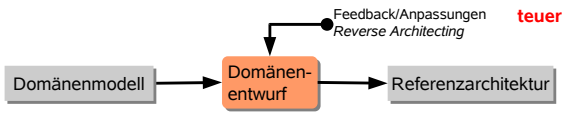
Betriebssystemtechnik

Operating System Engineering (OSE)

Architekturmodell: der Weg zur Klassen-Hierarchie

Domänenentwurf



- **Analyse**
  - Identifikation der Architekturtreiber, Mechanismen und Sichten
- **Modellierung**
  - **... ? ...**
  - Architekturdokumentation mit Variationspunkten
- **Evaluierung**
  - Erkennen von Risiken in den Entwurfsentscheidungen

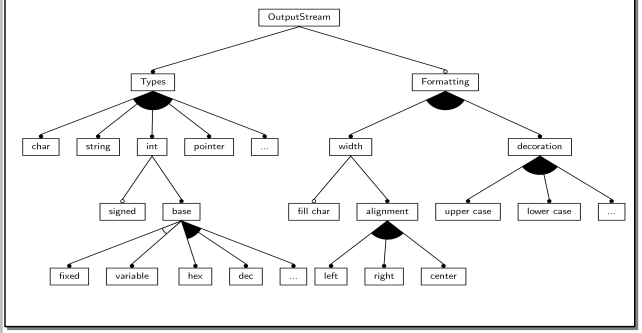
In Anlehnung an das Modell zur Architekturentwicklung aus [1].

Architektur-Modellierung

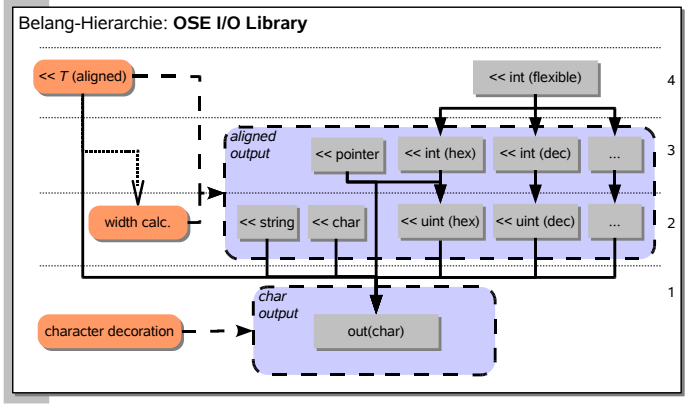
- ... am Beispiel eines BS-Subsystems
- bisher haben wir eine Belang-Hierarchie [1]
  - beschreibt funktionale Abhängigkeiten
  - erfasst quer schneidende Belange in den Funktionen
  - erlaubt Verfeinerungen von Funktionen durch eigene (Unter-) Belang-Hierarchien
- gesucht ist eine Modulstruktur
- Modularisierungsmechanismen werden i.d.R. durch die Programmiersprache bestimmt: hier AspectC++
  - Klassen, Methoden
  - **Aspekte**
  - (Templates)

Beispiel: I/O-Library

Merkmaldiagramm: **OSE I/O Library**

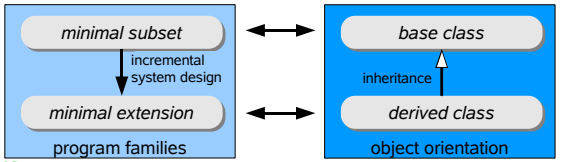


Beispiel: I/O-Library



OO vs. familienbasierter Entwurf

- Vererbung kann als Vehikel für die schrittweise Erweiterung genutzt werden

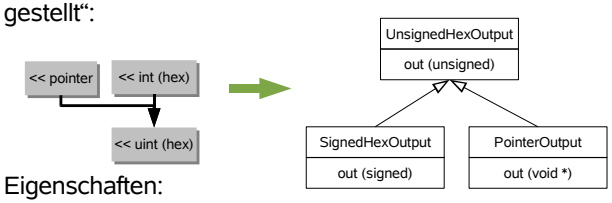


[2]

- **Pro**
  - funktionale Hierarchien lassen sich (relativ) leicht in eine Modulstruktur transformieren
- **Contra**
  - Entwurf wird leicht als schlechter OO-Entwurf missverstanden

Funktionen werden Klassen (1)

- die funktionale Hierarchie wird „einfach auf den Kopf gestellt“:

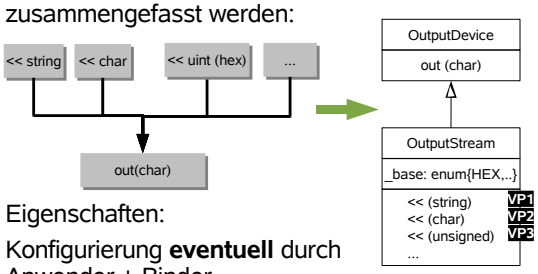


Eigenschaften:

- Konfigurierung durch Anwender + Binder
- eventuell Mehrfachvererbung
- Trennung des Zustands
- eignet sich für grobe Funktionen
  - Klassen ohne Attribute machen selten Sinn (siehe oben!)

Funktionen werden Klassen (2)

- Funktionen einer Ebene können auch zu einer Klasse zusammengefasst werden:

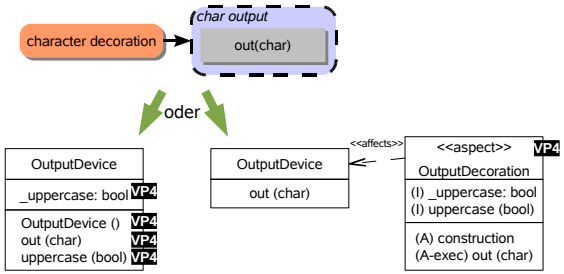


Eigenschaften:

- Konfigurierung **eventuell** durch Anwender + Binder
- gemeinsamer Zustand möglich
- für kleine Funktionen die angenehmere Schnittstelle

Quer schn. Bel. werden Aspekte (1)

- oft sogar sinnvoll, wenn nur eine Funktion betroffen ist:

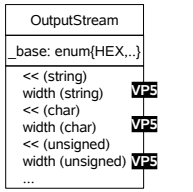


- der Aspekt entkoppelt Funktionen und erleichtert die Konfigurierung

Quer schn. Bel. werden Aspekte (2)

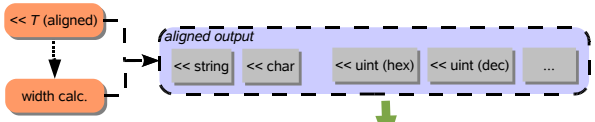
- in einigen Fällen ist die Modularisierung durch einen Aspekt nicht möglich:

- die Funktion width() ist hier so stark argumenttypabhängig, dass sie nicht durch einen Aspekt generisch realisierbar ist
- Konfigurierung **eventuell** durch Anwender + Binder

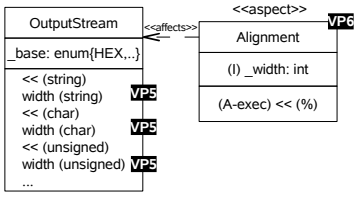


Quer schn. Bel. werden Aspekte (3)

- in der Regel klappt es aber doch:

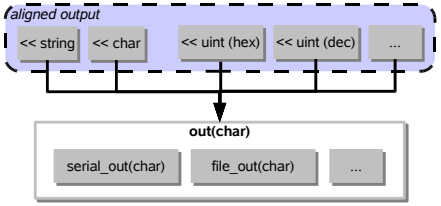


- der Advice-Code kann hier mit dem ersten Argument der Ausgabefunktion das überladene width() aufrufen und die Füllzeichen ausgeben



Konfigurierbare Funktionen

- ... machen eine gemeinsame Schnittstelle notwendig:



- wahlweise statisch gebunden, z.B. „Klassenalias“
- oder dynamisch gebunden, z.B. mit abstrakter Klasse

Zusammenfassung

- Grundregeln bei der Ableitung einer AO-Klassenhierarchie aus einer Belang-Hierarchie:
  - Erweiterungen (Ebenen) werden durch Vererbung ausgedrückt
  - Quer schneidende Belange können i.d.R. als Aspekte realisiert werden
  - Konfigurierbare Funktionen erfordern eine Schnittstellendefinition
    - Bindung kann statisch oder dynamisch erfolgen
    - das Hinzufügen abstrakter Schnittstellenklassen erfolgt immer anwendungsgetrieben, d.h. die Entwicklung beginnt **nicht** mit einer abstrakten Klasse wie es in OO-Entwürfen oft zu finden ist
- der Prozess vermeidet zwecks Konfigurierbarkeit zyklische Abhängigkeiten in der Modulstruktur

Ausblick

- Untersuchung verschiedener Techniken zur Umsetzung von Variabilität in der Implementierung der Komponenten
  - Stand der Kunst: Beispiel ECOS
  - Werkzeugbasierte Lösungen
  - Programmiersprachenbasierte Lösungen

Literatur

[1] O. Spinczyk. *Aspektorientierung und Programmfamilien im Betriebssystembau*. Dissertation, Otto-von-Guericke-Universität Magdeburg, 2002.

[2] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall, 1994. ISBN 0-13-183369-3.