

# **Vorlesung Software aus Komponenten**

## **2. Grundlagen**

apl. Prof. Dr. Hans-Gert Gräbe  
Wintersemester 2008/09

#### Definition Software-Komponente

Eine Software-Komponente ist eine Einheit der Komposition mit durch Kontrakt spezifizierten Schnittstellen und nur expliziten Kontext-Abhängigkeiten. Eine Software-Komponente kann unabhängig verteilt werden und zur Komposition durch Dritte verwendet werden.

- Definition wurde erstmals so 1996 auf der „European Conf. on OO Programming“ gegeben [Szyperski, Pfister]
- technische Seite: Unabhängigkeit, Schnittstellen-Kontrakt, Zusammenbau
- soziale Seite: Dritte, Verteilung
- Diese Verbindung ist typisch für den Komponentenbegriff nicht nur im Software-Bereich

## Schnittstellen-Kontrakt

- Muss die Verwendung der Komponente in einem Produktiv-System genau beschreiben
- Aspekte:
  - Schnittstellen im engeren Sinne
  - Entpackung, Konfiguration, Installation der Komponente
  - Instanziierung und Beschreibung des Verhaltens der durch die Komponente erzeugbaren Objekte durch ihre Schnittstellen (Laufzeitverhalten)
  - Beschreibung der Ressourcenanforderungen der Komponente sowie der Anforderung an die Lokalisierungs-Umgebung
    - auch als Kontext-Abhängigkeit bezeichnet
  - enthält:
    - Komponenten-Modell = Spezifikation der Kompositions-Regeln
    - Komponenten-Plattform = Spezifikation der Regeln für Entpackung, Installation und Aktivierung von Komponenten

- Schnittstellen-Spezifikation als Kontrakt zwischen
  - Nutzer der **Funktionalität** einer Schnittstelle und
  - Anbieter der **Implementierung** dieser Schnittstelle
- verbreiteter Zugang auf technischer Ebene: durch Vor- und Nachbedingungen (Hoare-Kalkül:  $\{V\} P \{N\}$ )
  - Nutzer sichert die Vorbedingungen V
  - Anbieter sichert dann Nachbedingung N
  - Problem: Sichert funktionale Eigenschaften, aber weder Performanz von P noch Termination überhaupt
- heute üblich: auch nicht-funktionale Aspekte im Kontrakt erfassen
  - Beispiel: Service Level Agreement
    - enthält Qualitätsaussagen für den Betrieb wie Verfügbarkeit, Fehlerrate, Datensicherheit etc.
  - Konsequenzen im Einsatz sind ähnlich gravierend wie funktionale Fehler

- „undokumentierte Features“
  - Auf Komponenten-Verhalten kann jenseits der Spezifikation auch aus Beobachtung des laufenden Betriebs geschlossen werden
  - typisches Ergebnis eines „Debugging“-Prozesses bei Fehlersuche
  - kleine Fehler sind ökonomischer auf der Nutzerseite als auf der Anbieterseite zu beheben
  - Open-Source-Ansatz

## Direkte und indirekte Schnittstellen

- Direkte Schnittstelle: Schnittstelle der Komponente selbst
  - meist prozeduraler Natur
- Indirekte Schnittstelle: Schnittstelle von Objekten, die in der Komponente erzeugt werden
  - meist objektorientierter Natur
- Vereinheitlichung durch Einführung eines statischen Objekts möglich
  - typischer Ansatz von OO Sprachkonzepten
- Überladung indirekter Schnittstellen und späte Bindung
  - Provider des Dienstes hängt vom Objekt ab
  - Derselbe Dienst kann über dieselbe Schnittstelle innerhalb desselben Komponenten-Kontexts von unterschiedlichen Anbietern kommen

## Schnittstellen und Versionen

- Problem: Schnittstellen können ihr Verhalten zwischen Versionen wechseln
- Management traditionell über Versionsnummern
  - Komponente als unteilbare Einheit => Versionsnummern nur für ganze Komponenten
- Versions-Information in Import- und Exportschnittstellen
  - direkte Schnittstellen: Abfrage zur Bindungszeit, also (nur) vor dem ersten Schnittstellenaufruf
  - indirekte Schnittstellen: Abfrage vor jedem Aufruf erforderlich
    - Alternative: Integration ins Management der Objektidentität
- Problem der Versions-Information, wenn eine Objektreferenz Komponentengrenzen überschreitet
  - Objekt bietet eigene Dienste an
    - etwa Rückgabe in einem bestimmten Format
  - Objekt nutzt Dienste anderer => dynamische Versionskontrolle

- Schnittstellenversionen müssen klar als kompatibel oder klar als veraltet (deprecated) deklariert werden können
- Ansatz der unveränderlichen Schnittstellen-Spezifikation (immutable interfaces)
  - Neue Versionen nur als neue Schnittstellen
  - Veraltete Schnittstellen werden einfach nicht mehr unterstützt
  - Beispiel: COM = Component Object Model von MicroSoft
- Veränderbare Schnittstellen-Spezifikationen:
  - klares Kompatibilitäts-Konzept erforderlich
  - Installation verschiedener Komponentenversionen in derselben Umgebung kann erforderlich sein.
  - Unterscheidung zwischen Komponenten, die immer in der aktuellsten Version verwendet werden können und Komponenten, die nur in der ursprünglich installierten Version verwendet werden können
    - wird im Rahmen der CLR = Common Language Runtime verfolgt

## Vererbung als Prinzip

- Drei wesentliche Facetten
  - Subklassen: Vererbung von Implementations-Fragmenten
    - Implementations-Vererbung
    - Java: *extends*
  - Subtypen: Vererbung von Kontrakt-Fragmenten
    - Schnittstellen-Vererbung
    - Java: *implements*
  - Substituierbarkeit: Vererbung funktionaler Eigenschaften
- Mehrfach-Vererbung
  - Kein Problem auf Schnittstellenebene
  - Diamanten-Problem auf Implementationsebene
    - auf der Ebene der Zustände (Attribute – Referenz oder Kopie?)
      - Referenz bricht Kapselung
    - auf der Ebene der Methoden

## Das Problem der fragilen Basisklasse

- Fragestellung: Was passiert in einer Vererbungsrelation, wenn die Basisklasse durch eine neue Version ersetzt wird?
  - syntaktische Dimension: Wie sieht es mit der Binärkompatibilität von neuer Basisklasse und alten Kindklassen aus?
    - Muss die Kindklasse recompiliert werden?
    - Wenn nur Methoden vererbt werden: im Prinzip nein
    - Selbst bei Verschiebungen in der Vererbungshierarchie (Restrukturierung) oder Schnittstellen-Erweiterungen nicht
    - Grund: Methodenbindung erfolgt zur Laufzeit über die Dispatch-Tabellen der Klassen
  - semantische Dimension: Die Implementierung der Subklasse nimmt Bezug auf implementatorische (semantische) Details der Basisklasse
    - mit einer neuen Version der Basisklasse kann diese Voraussetzung hinfällig sein.
    - auch durch Recompilieren nicht aus der Welt zu schaffen

Basisklasse implementiert read/write auf abstrakter Ebene

```
abstract class Text {  
    private char[] text = new char[1000]; // Textbuffer  
    private int used = 0; // Position des letzten Textzeichens  
    private int caret = 0; // Position des Cursors  
  
    void setCaret(int pos) { caret=pos; }  
    int caretPos() { return caret; }  
    ...  
    void write(int pos, char ch) {  
        for (int i=used; i>pos; i--) { text[i]=text[i-1]; }  
        used++; text[pos]=ch;  
        if (caretPos()>=pos) setCaret(caret+1);  
    }  
    abstract int posToX(int pos);  
    abstract int posToY(int pos);  
    abstract int posFromCoords(int x, int y);  
}
```

Subklasse implementiert View mit Cursor

```
class SimpleText extends Text {  
    private int cacheX = 0; private int cacheY = 0;  
  
    void setCaret(int pos) { // überschriebene Methode  
        int old = caretPos();  
        if (old != pos) { hideCaret(); super.setCaret(pos); showCaret(); }  
    }  
    int posToX(int pos) {...}  
    int posToY(int pos) {...}  
    int posFromCoords(int x, int y) {...}  
    void hideCaret() { ... }  
    void showCaret() { ... }  
}
```

Neue Version der Klasse Text:

```
abstract class Text {  
    ...  
    void write(int pos, char ch) {  
        for (int i=used; i>pos; i--) { text[i]=text[i-1]; }  
        used++; text[pos]=ch;  
        if (caret >= pos) caret++;  
    }  
}
```

Effekt: Nach *write* steht der grafische Cursor an der falschen Position, weil die Implementierung von *SimpleText* sich darauf verlassen hat, dass *Text* die Variable *caret* stets nur über *setCaret* manipuliert.

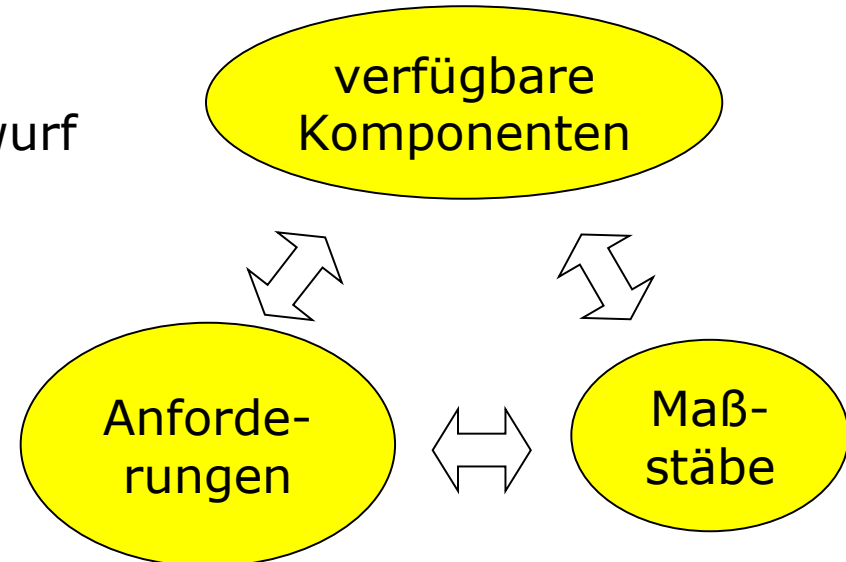
Wechsel der inneren Implementierung des Anbieters kann den Nutzer korrumpieren, ohne den Kontrakt zu verletzen.

### Komponenten abgrenzen

Problem: Wie ist ein kompletter Entwurf in Komponenten zu partitionieren?

Prinzipien:

- Modularität und Kapselung
- Abhängigkeiten zwischen K. sind expliziert
- Mehrere Abstraktionsebenen, hierarchische Strukturierung
- natürliche Zuordnung von Verantwortlichkeiten
- Migrations-Erfordernisse vorab berücksichtigen



- Wie „fett“ soll eine Komponente sein?
  - optimal, aber unreal: „richtige Schnittstellenmenge“ und keine Kontext-Abhängigkeit
  - maximal: „fette“ Komponente, die alle benötigten Dienste mitbringt (=Applikation, grobkörnig)
  - minimal: Auslagern aller bis auf die zentrale Funktionalität (=Klasse, feinkörnig)
    - „Maximizing reuse minimizes use.“
    - Grund: Explodierende Kontext-Abhängigkeit
    - würde nur unter statischen Entwicklungsbedingungen funktionieren
    - Beispiel: Linux-Probleme mit Bibliotheksversionen
  - praktisch ist hier ein je ausgewogenes Mittel zu finden
- Je detaillierter Normierung und Standardisierung, desto schlankere Komponenten sind möglich
  - Standardisierung ist in vertikalen Marktsegmenten (funktional) eher möglich als in horizontalen, aber wegen der geringen Marktgröße schwieriger

## Typische Betrachtungsansätze

(führen zu unterschiedlicher Granularität)

- **K. als Einheit der Abstraktion**
  - Ansatz: white box → black box
  - Entwurfsexpertise kapseln (design expertise ready for use)
  - Theoretische Einschränkung der Vielfalt in der aktuellen Ebene ist Basis für größere Vielfalt in der nächsthöheren Ebene
- **K. als Einheit der Kostenrechnung**
  - wichtig in größeren industriellen Kontexten, um Projektentwicklungskosten verfolgen zu können

- **K. als Einheit des Managements**

- oft zu klein, Management auf der Ebene von Subsystemen, die mehrere Komponenten zusammenfassen
  - etwa auf der Ebene der Server

- **K. als Einheit der Analyse**

- Analyse an vielen Stellen im Komponentenlebenszyklus erforderlich (Spezifikationsprüfung, Tests, Re-Engineering)
- Kopplung zwischen Einheiten bestimmt, wie weit eine individuelle Analyse zweckmäßig bzw. überhaupt möglich ist
- Regel: Einheiten für Analyse so klein wie möglich
- Regel: streng statische hierarchische Grenzen (Moduln, Subsysteme) erleichtern die Analyse

- **K. als unabhängig compilierbare Einheit**
  - Compilierbarkeit und Analyse sind eng verbunden
    - white box: Analyse
    - black box: Compilierbarkeit
  - sinnvolle Einheiten: Moduln oder Klassen
  - Globale Optimierung?
    - Antwort 1: Einheiten möglichst groß wählen
    - Antwort 2: Optimierung zwischen Komponenten, vielleicht sogar erst nach der Lokalisierung
      - muss im Komponentenkonzept und der Komponentenbeschreibung verankert sein
- **K. als Auslieferungseinheit**
  - Bündel der technischen und wirtschaftlichen Aspekte
  - Management (Service, Wartung, Schulung, Updates, ...) treibt den Preis in die Höhe
    - betriebswirtschaftliche Bedeutung jenseits der (geringen) Replikationskosten

- **K. als Packungs-Einheit**

- Entpackung (deployment) = Prozess der Vorbereitung der Komponente auf den Einsatz in einer speziellen Umgebung (Lokalisierung)
  - wurde lange nicht als separater Schritt betrachtet
  - Prozess der Anbindung an eine spezielle Komponentenplattform
- Konfiguration = Einstellung spezieller Eigenschaften der Komponente für den konkreten Einsatz
- Installation = plattformspezifische Aktivität, mit der eine entpackte Komponente für die Nutzung in einer speziellen Hardware-Konfiguration verfügbar gemacht wird, die von der Plattform unterstützt wird.
  - Zeit, in der auch kritische Tests ausgeführt werden, die vor dem eigentlichen Betrieb erfolgen müssen (etwa Integritätstests)
- für alle drei Aktivitäten müssen entsprechende Beschreibungen erstellt werden

- **K. als Einheit der (Auseinandersetzung um) Fehlersuche**
  - Problem: Was ist (u.a. wer haftet?), wenn ein aus Komponenten zusammengebautes Produktivsystem fehlerhaft arbeitet?
  - Problem der Lokalisierung von Fehlern (und damit Verantwortlichkeiten)
    - besonders schwierig wenn Objektreferenzen die Komponentengrenzen verlassen
  - vitale Regel: Fehler müssen in den verursachenden Komponenten bleiben (bug containment)
    - typische nicht-lokalisierbare Fehler: Speicherzugriffsfehler
  - Folge: Ausnahmebehandlungen müssen in der Regel innerhalb einer Komponente bleiben
    - Ausnahmen davon sind im Komponentenkontrakt zu fixieren
- ⊞ ➔ Komponente als Einheit der Fehlerbehandlung

- **K. als Einheit der Erweiterung**

- Beispiel: K. implementiert eine konkrete (standardisierte) Schnittstelle
- Objekte in einer Erweiterungshierarchie sind gewöhnlich enger gekoppelt als andere
  - „friends“-Mechanismus, „protected“-Schnittstelle
  - sind Ersatz für Mechanismen der Kapselung mehrerer Objekte
- unabhängige Erweiterungen und deren Koexistenz
- eine Einheit der Analyse darf nicht in mehrere Erweiterungseinheiten zerlegt werden
  - gemeinsame Analyse hat oft kontextuelle Abhängigkeiten zwischen den Teilen zur Folge

- **K. als Einheiten der Instanziierung**

- Einheit der Instanziierung sind Objekte
- also: kein sinnvolles Kriterium für Komponentenabgrenzung

- **K. als Einheit des Ladens**

- oft wird beim ersten Einsatz einer Funktionen erst entsprechender Code geladen.
- benötigt Mechanismen des dynamischen Ladens (etwa DLL)
- typisch werden dabei gleich mehrere zusammenhängende Klassen geladen
  - ⊞ ➔ Komponente als Einheit des Ladens sinnvoll
- Kontrolle der Version
  - Problem der fragilen Basisklasse
  - Maximale Flexibilität, wenn Versionskontrolle auf der Ebene von Schnittstellen oder sogar Methoden
  - Java: Versions-Check erst zur Laufzeit
    - ist problematisch, da Inkompatibilitäten erst mitten in der Programmausführung entdeckt werden
- Kontrolle der Erfüllbarkeit der Importrelationen
- Kontrolle von Namenskollisionen
  - Lösung: Hierarchisches Namensraum-Konzept
  - Problem der Namenskollision: Versionen derselben Komponente

- K. als Einheit des Ladens (Fortsetzung)
  - Konsistenz bereits geladener Komponenten muss erhalten bleiben
    - Konsistenz muss dazu lokale Eigenschaft sein
    - schließt z.B. globale Typprüfungsansätze aus
- **K. als Einheit der Lokalität**
  - Problem im Kontext verteilter Systeme: was befindet sich (lokal) auf welchem Rechner
    - typisch sind hierarchische Konzepte des Clusters von Rechnern
      - SAN, LAN, WAN, Internet (und weitere Zwischenstufen)
      - unterschiedliche Kommunikationszeiten und -kosten
  - Tradeoff: minimale Kommunikationskosten vs. maximale Ressourcennutzung
  - hohe Kopplung zwischen Objekten aus derselben Komponente
    - ⊞ ➔ Komponente sollte nicht auf verschiedene Prozesse oder Maschinen verteilt sein
    - Bündelung von Zugriffen auf Objekte über Prozessgrenzen hinweg
      - ein komplexer statt mehrerer einfacher Zugriffe