

# **Vorlesung Software aus Komponenten**

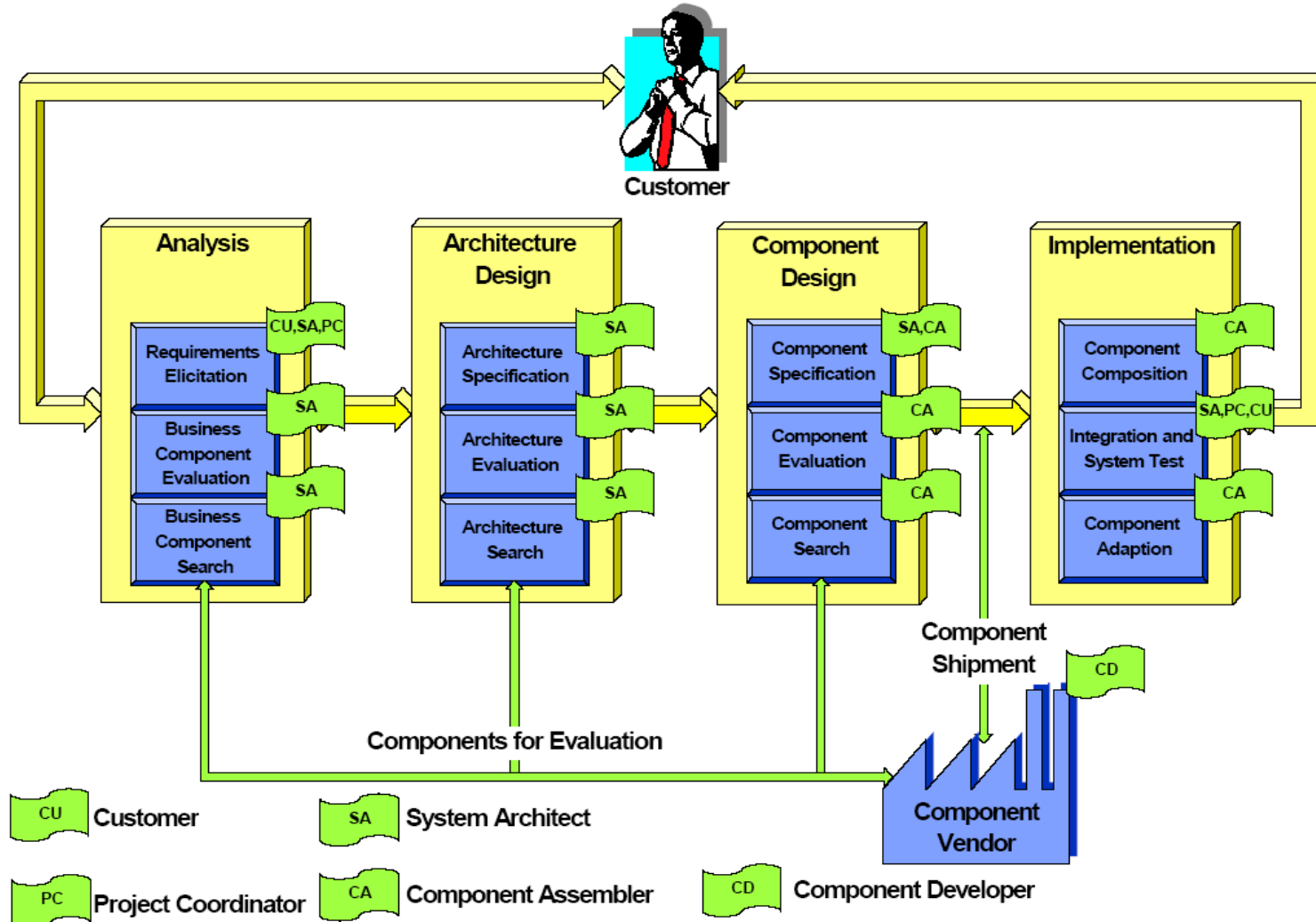
## **1. Komponenten – Markt – Standards**

Prof. Dr. Hans-Gert Gräbe  
Wintersemester 2011/12

# 1.4. Komponentenentwicklung

## Entwicklungszyklus

### Entwicklungszyklus für Komponenten-Software



Quelle: /Sihling 1998/

## ***Komponententechnologie:***

Mehrstufiger modularer Softwareaufbau aus vorgefertigten Komponenten, in denen Funktionalität so gekapselt ist, dass diese auf einheitliche Weise wiederverwendet werden kann

## ***Anforderungen:***

- formal fundiertes **Komponentenkonzept** als Basis
- **Beschreibungstechniken** für derartige Komponenten
- Entwicklung eines **Prozessmodells** zur Entwicklung, Verwaltung und Zusammensetzung von Komponenten
  - Unterstützung der Zuweisung verschiedener Rollen
- **Werkzeuge**, welche die Beschreibung und das Prozessmodell unterstützen
  - zur Systemgenerierung selbst
  - zur Dokumentation
  - zur Verifikation und Sicherung wichtiger und kritischer Systemeigenschaften

# **Vorlesung Software aus Komponenten**

## **2. Grundlagen**

Prof. Dr. Hans-Gert Gräbe  
Wintersemester 2011/12

#### Was sind Komponenten und was nicht?

- „Component Software – beyond object oriented programming“
- Problem der Abgrenzung der Begriffe „Objekt“ und „Komponente“
  - synonym in manchen (Kon)texten
  - ex. auch Begriffe wie „Komponenten-Objekt“
- Objektbegriff: Instanz einer Klasse (Java) oder Clon eines Prototyp-Objekts (Factory-Pattern)
- Gemeinsamkeiten:
  - Dienste werden über Schnittstellen angesprochen
  - Schnittstellen nach Beschreibungsregeln typisiert und kategorisiert
  - Beschreibung nach Mustern (Syntax) und innerhalb von Frameworks (Semantik)
- weitere Verwirrung durch Sprachdesigner:
  - Namensräume, Moduln, Packages

- Notwendigkeit, in diesem Universum von Namen und Konzepten **aufzuräumen**, Begriffe an definierte **Bedeutungen** zu binden und in ein gewisses **Ordnungsschema** zu bringen.
- Ansatz: Bedeutungen von Begriffen durch Angabe charakteristischer Eigenschaften fixieren

### Komponentendefinition

Charakteristische Eigenschaften einer Komponente:

1. Einheit unabhängiger Packung
2. Einheit der Komposition durch Dritte
3. ohne (extern beobachtbaren) Zustand

### Folgerungen aus der Definition

#### Einheit unabhängiger Verteilbarkeit

- separiert von Umgebung und anderen Komponenten
- Kapselung charakteristischer Merkmale

#### Einheit unabhängiger Verteilbarkeit

- nie in Teilen zu verteilen

#### Komposition durch Dritte

- Verwendung ohne Kenntnis konstruktiver Details

#### Komposition durch Dritte

- genügend selbstabgeschlossen
- klare Spezifikation der Anforderungen und Angebote

#### zustandslos

- ohne diese Forderung hätten keine zwei Installationen „derselben“ Komponente garantiert gleiche Eigenschaften
- Zustand spielt sich ausschließlich auf der Ebene der Objekte ab

### Zustandslos

- Kann nicht von Kopien unterschieden werden
  - Ausnahme: Zustandsparameter jenseits von Funktionalität
- Intern verwendete Zustände (oft aus Gründen der Leistungsfähigkeit) dürfen von außen nicht sichtbar sein
  - Beispiel: Cache
- Nicht sinnvoll, in einer Umgebung mehrere Instanzen einer Komponente zu halten
  - Achtung Konfusion vermeiden! Beispiel Datenbank. Unterscheide zwischen Datenbank-Server-Programm (Komponente) und darauf laufender Datenbank als „Instanz“ des Datenbank-Konzepts.
  - Diese Unterscheidung zwischen dem (fest verdrahteten) „Plan“ ( = Komponente) und den (sich im Laufe der Zeit ändernden) „Instanzen“
  - Analog der Unterscheidung zwischen Objekten und Objektzuständen

Komponenten sind heute praktisch meist schwergewichtige Einheiten mit genau einer Instanz pro System



## Objektdefinition

Charakteristische Eigenschaften eines Objekts:

1. Einheit der Instanziierung mit (eindeutiger) Identität
2. kann (extern beobachtbaren) Zustand besitzen
3. kapselt Zustand und Verhalten

Folgerungen:

Einheit der Instanziierung

- partielle Instanziierung nicht möglich

#### Eindeutige Identität

- Zustand für jedes Objekt individuell, kann sich im Laufe des Objekt-Lebenszyklus ändern
- einzig garantiert persistente Eigenschaft eines Objekts ist dessen abstrakte Identität
- Werner Brösels Auto bleibt W.B. Auto, auch wenn W.B. im Laufe der Zeit alle Teile daran ausgetauscht hat (und das Auto von seinem Original nichts mehr hat als eben diese abstrakte Identität)

#### Einheit der Instanziierung

- Plan muss Zustandsraum, Anfangszustand und (anfängliches) Verhalten eines neuen Objekts beschreiben
- Plan muss vor der Existenz der Objekte existieren
- Plan kann explizit sein = Klasse
- Plan kann implizit sein = Prototyp-Objekt
- Initialzustand muss valide sein, kann aber von weiteren Parametern abhängen
  - Code zur Erzeugung kann statisch sein = Konstruktor
  - Code zur Erzeugung kann selbst Objekt sein = factory object
  - Objekte können von anderen Objekten erzeugt werden = factory method

## Komponenten und Objekte

- Komponenten entfalten ihre Wirkung in der Regel über Objekte
  - Eine Komponente besteht also meist aus mehreren Klassen oder unveränderlichen Prototyp-Objekten sowie weiteren Komponenten-Ressourcen
- Komponenten können auch vollkommen anders realisiert sein
- Export von Objekt(referenzen) bedeutet nicht, dass es intern auch OO zugeht und kann (allein über Objektreferenzen) von außen auch nicht erforscht werden
- Komponenten und Klassen
  - Komponenten können mehrere Klassen umfassen
  - eine Klasse kann nicht über mehrere Komponenten verteilt sein

## Komponenten und Vererbung

- Vererbung ist über Komponentengrenzen hinweg möglich
  - muss aber in der Komponenten-Schnittstelle über eine Import-Deklaration explizit verankert sein
- Vererbung als Prinzip – drei wesentliche Facetten
  - Subtypen: Vererbung von Kontrakt-Fragmenten
    - Schnittstellen-Vererbung, Java: *implements*
    - wesentliche Technik, um Korrektheit über Komponentengrenzen hinweg zu garantieren
    - gängiger Weg zur Operationalisierung von Standards
  - Subklassen: Vererbung von Implementations-Fragmenten
    - Implementations-Vererbung, Java: *extends*
    - relativ schwierige Problematik über Komponentengrenzen hinweg
    - Problem der fragilen Basisklasse
  - Substituierbarkeit: Vererbung funktionaler Eigenschaften

## 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 (mglw. implizit) Bezug auf implementatorische (semantische) Details der Basisklasse
    - Mit einer neuen Version der Basisklasse kann diese Voraussetzung hinfällig sein.
    - Das ist 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]; // Textpuffer [0..(used-1)]  
    private int used = 0; // Position nach dem letzten Textzeichen  
    private int caret = 0; // Position des Cursors  
  
    void setCaret(int pos) { caret=pos; }  
    int caretPos() { return caret; }  
    ...  
    void write(int pos, char ch) { // füge Zeichen ch an Stelle pos im Puffer ein  
        for (int i=used+1; 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() { ... }  
    // Methode write wird unverändert geerbt und setzt den Cursor richtig.  
}
```

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++; // Position wird aktualisiert ohne setCaret!  
    }  
}
```

**Effekt:** *write* setzt nun den grafischen Cursor nicht mehr. Dieser steht 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.



## Moduln

Komponenten sind eher Moduln ähnlich. Wo sind die Unterschiede?

Typische Eigenschaften von Moduln:

- separate Compilierbarkeit
- Mechanismen zur Typprüfung über Modulgrenzen hinweg
- Historisch Strukturierungsinstrument alternativ zu Klassen, nicht instanziiierbar
- Behauptung [Meyer 1988]: Klassen sind das bessere Modulkonzept, ein separates Modulkonzept wird nicht benötigt, denn Klassen sind mehr: Klasse = Schnittstelle + Polymorphie + Vererbung

Moderne Sprachen (Modula-3, Oberon, C#) trennen das dennoch wieder

- Moduln als semantische Strukturierungseinheit zur Code-Zusammenfassung **oberhalb** von Klassen
  - C# und .NET: Assemblies
  - Java Packages liefert als vergleichbares Konzept funktional deutlich weniger

Im Gegensatz zu Klassen können Moduln die Grundlage für minimale Komponenten bilden

- Beispiel: math-Bibliotheken
- sind eher funktionaler als objekt-orientierter Natur

Moduln unterstützen keine Zuordnung persistenter unveränderlicher Ressourcen (jenseits hart im Code „verdrahteter“ Konstanten)

- Komponenten werden mit lokal verfügbaren Ressourcen zu einer „lokalen Komponente“ konfiguriert
- die Beziehung zwischen einer Komponente und ihren Lokalisierungen ist komponenten-technologisch wichtig, im Modulkontext ausgeblendet

Nicht jeder Modul geht als Komponente durch

- erlaubt: globale Variable, statische Abhängigkeit von Implementierungen in anderen Moduln (= zustandsbehaftet)

Zusammenfassung: Modularität ist eine Voraussetzung für Komponenten-Technologie

## Whitebox und Blackbox

Thema: Sichtbarkeit der Implementierung hinter der Schnittstelle

Blackbox: Der Nutzer weiß nichts über die Interna

- Konzept der Komponente als Menge von Schnittstellen und deren Spezifikation
  - Bsp: API-Programmierung

Whitebox: Die Schnittstelle kontrolliert die Zugriffsrechte, Interna sind aber prinzipiell bekannt

- Konzept der Komponente als Software-Fragment
  - schwache Form: durch Studium zusätzliche Informationen über das interne Verhalten der Komponente bekommen (glassbox)
  - starke Form: Implementations-Vererbung
- Entwickler studieren die Implementierung
  - Problem des Release-Wechsels
- Viele Klassenbibliotheken und Frameworks fallen unter diese Kategorie