

# **Vorlesung Software aus Komponenten**

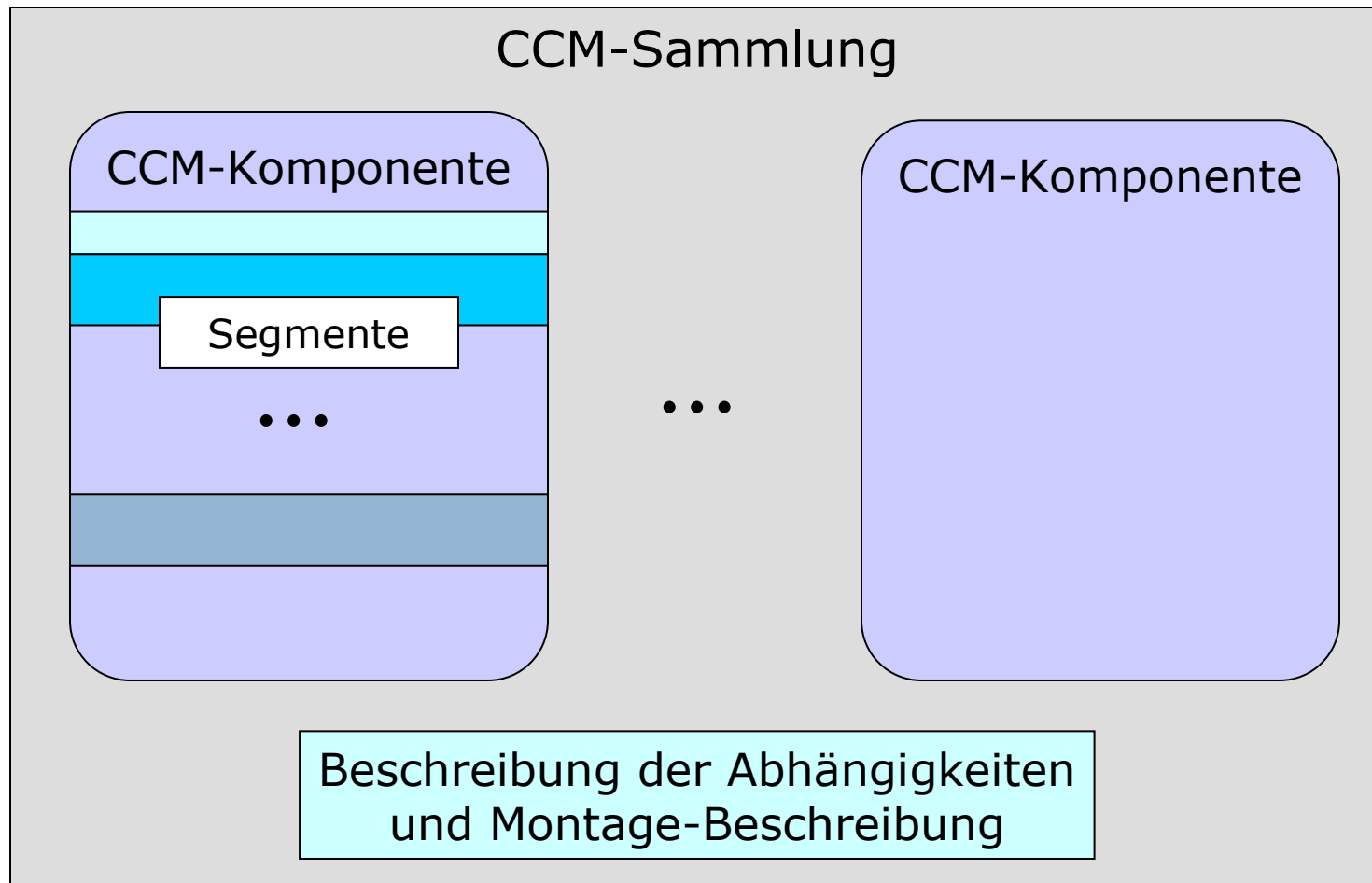
## **3. Komponenten-Modelle**

Prof. Dr. Hans-Gert Gräbe  
Wintersemester 2006/07

### Das CORBA Komponentenmodell (CCM)

- mit CORBA 3.0 endgültig spezifiziert
- Ambitionierte (logische) Erweiterung des EJB-Ansatzes
- CCM-**Anwendung** besteht aus CCM-**Komponenten**
  - EJB erfüllen die CCM-Komponenten-Spezifikation
- CCM-Komponenten sind in **Komponentenpaketen** zusammengefasst
- CCM-**Sammlungen** (CCM assemblies) enthalten Komponentenpakete zusammen mit einer **Beschreibung** der Abhängigkeiten und der Montage-Beschreibung im XML-Format
- Eine CCM-Komponente kann aus mehreren **Segmenten** bestehen
  - CCM-**Laufzeitumgebungen** laden Anwendungen segmentweise
- CCM-Anwendungen laufen nur mit CORBA-3-konformen ORBs
  - wird auch auf der Client-Seite benötigt, wenn die ganze CCM-Funktionalität (etwa Navigation) ausgenutzt werden soll
  - CCM-Standard unterstützt aber abgerüstete Klienten auf pre-CORBA-3-Plattformen (component-unaware clients)

### Grundstruktur einer CCM-Sammlung



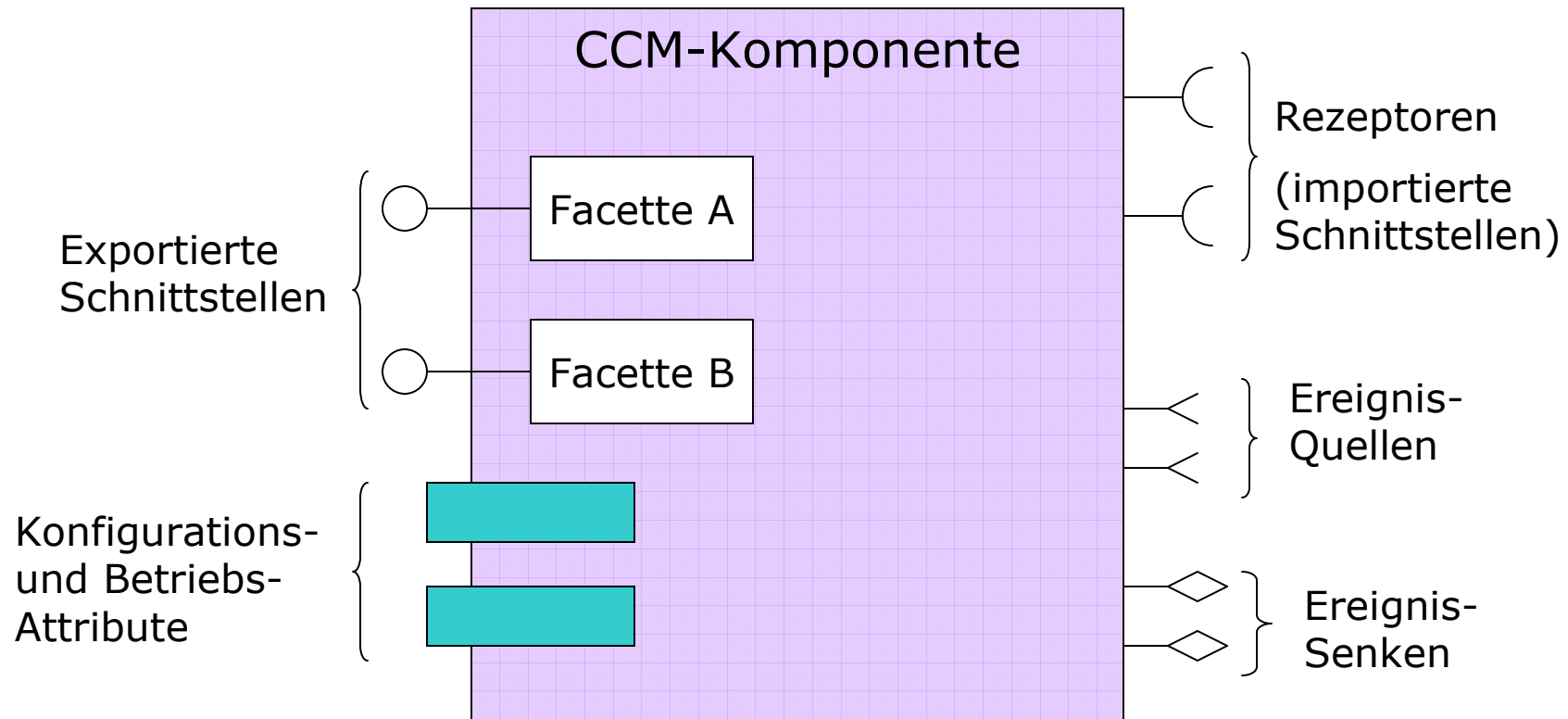
## CCM-Kategorien

- CCM-Komponenten werden (ähnlich EJB) in **Kategorien** eingeteilt
  - **Service-Komponenten**
    - Instanzen sind Aufrufen zugeordnet und speichern keine Zustände über Aufrufgrenzen hinweg
  - **Session-Komponenten** ( = stateful session EJB)
    - Verwaltung des Zustands innerhalb eines Transaktionszyklus (transactional session)
  - **Entity-Komponenten** ( = entity EJB)
    - Instanzen haben persistenten Zustand, entsprechen Datenbankeinträgen
    - können über Primärschlüssel aus einer Datenbank gefunden werden
  - **Prozess-Komponenten**
    - persistent, Lebensdauer an die des Prozesses gebunden, der bedient wird
- CCM-Anwendung enthält deklarative Informationen über Komponentenkategorien und Komponentenaufgaben

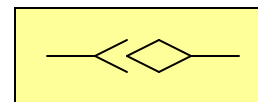
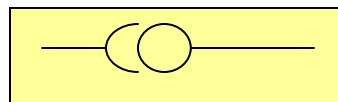
# 3.7. Das CORBA-Komponentenmodell

## Aufbau einer CCM-Komponente

### Aufbau einer CCM-Komponente



Kopplung:



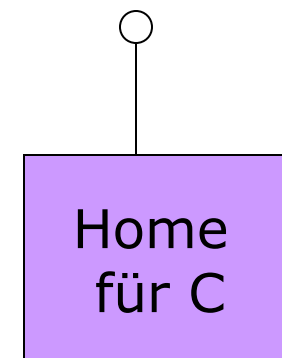
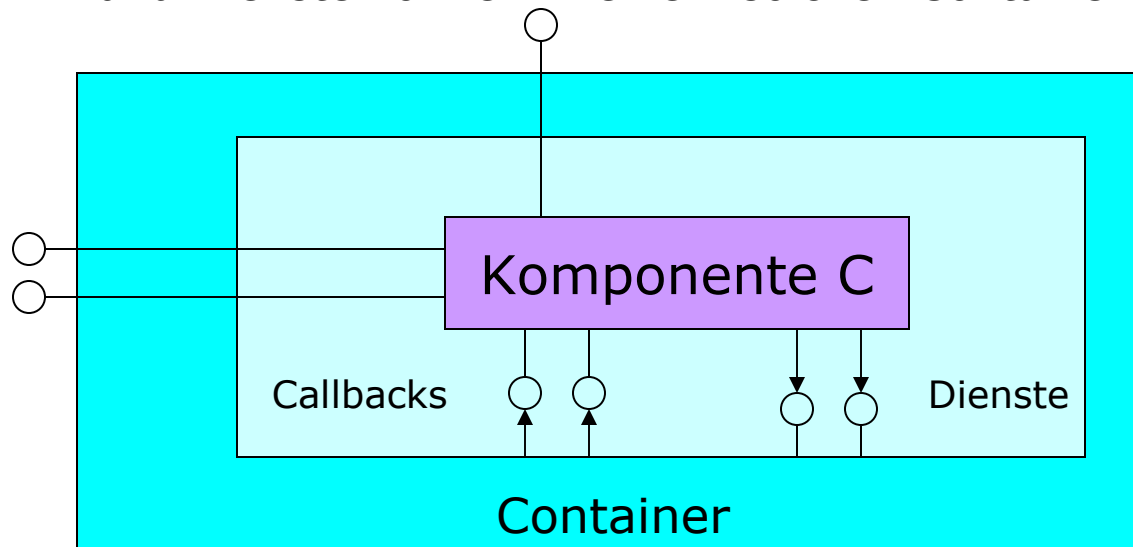
### Ports von CCM-Komponenten

- **Facetten** (facets)
  - exportierte Schnittstelle, gewöhnlich einem Teilobjekt der Komponente zugeordnet
- **Rezeptoren** (receptables)
  - importierte Schnittstellen, intern Referenzen auf externe Objekte, die zum Komponentenbetrieb benötigt werden
  - connect / disconnect Operationen
  - können explizit in der Montage-Beschreibung gefordert oder zur Laufzeit eingebunden werden
- **Ereignisquellen** (event sources) und **Ereignissenken** (event sinks)
  - durch Ereigniskanäle zu verbindende Ports
- **Primärschlüssel** (nur Entity-Komponenten)
- **Konfigurations-** und **Betriebs-Attribute**
  - benannte Werte, die über **Zugriffsfunktionen** (accessor) oder **Modifizierer** (mutator) nach außen sichtbar sind

- **Home-Schnittstelle**, über welche die Komponenten-Factory erreicht werden kann
  - in der Komponenten-Klasse implementiert
  - also Komponentenbegriff verschieden von dem in der Vorlesung
  - Management des Lebenszyklus von Komponenten-Instanzen
- Spezielle Facette **E-Schnittstelle** (equivalent interface), über die zwischen den Facetten der Komponente navigiert werden kann
  - analog der IUnknown-Schnittstelle im COM-Konzept
  - Clienten müssen CORBA-3 unterstützen, um diese Navigationsmöglichkeiten auszunutzen
- **Konfigurations-Schnittstelle** (configuration interface)
  - Unterstützung der initialen Konfiguration neuer Komponenten
  - spezielles call-Signal schließt die Konfigurationsphase ab
  - erst danach sind Aufrufe der operationalen Schnittstellen möglich, Aufrufe der Konfigurations-Schnittstelle dagegen untersagt

### CCM-Container

- CORBA 3 definiert ein **Komponenten-Implementierungs-Gerüst** (component implementation framework, CIF)
  - Generatoren erzeugen aus Eingaben im **CIDL-Format** (component implementation description language) Code, der den Komponentencode ergänzt
- Jede Komponenten-Instanz ist in einem **CCM-Container** untergebracht, über den die Anbindung der Facetten und Rezeptoren erfolgt. Rezeptoren und Dienste können in einem solchen Container per Callback gebunden sein.





## 3.7. Das CORBA-Komponentenmodell CCM-Container

- Navigation zwischen den Schnittstellen aller Komponenten im Container
  - Container hat Kontrolle über exportierte und importierte Schnittstellen seiner Komponenten
  - kann nur von CORBA-3-konforme Clients genutzt werden
- CCM-Container ist ein spezieller POA mit vorgefertigten Basisdiensten (pre-packaged object services)
  - **Transaktionsdienst:** durch Container oder selbst
    - Komponente: Beschreibung enthält die Transaktionsanforderungen (supported, required, required new, not supported)
    - Container: Ausführung der Transaktionen entsprechend der Spezifikation der einzelnen Komponenten
  - **Persistenzdienst:** durch Container oder selbst
    - Komponente: Beschreibung der Anforderungen im PSDL-Format (persistent state description language)
  - **Sicherheitsdienst:** Zugriffsrechte können im CIDL-Format beschrieben und durch den Container geprüft werden
  - **Benachrichtigungsdienst:** Aufbau und Verwaltung von Ereigniskanälen

### Zwei grundlegende Herangehensweisen

- eng gekoppelte Architektur (J2EE, CORBA, OLE und CLR)
  - Zusammenschalten der Rechner zu einem „verteilten Betriebssystem“ als Infrastruktur, in der Objektinstanzen ausgetauscht werden, in denen Zustand und Funktionalität des Gesamtsystems lokal gespeichert sind.
  - fein granulares Konzept, Technik der Interaktion steht im Fokus
  - Erweiterung objektorientierter Ansätze von einer Einzelplatzanwendung auf eine verteilte Umgebung
  - grundlegendes Konzept: RPC und dessen Verallgemeinerungen
- lose gekoppelte Architektur (Webservices)
  - hohe Autonomie der Rechner, die nachrichtengesteuert gegenseitige „Dienste“ erbringen
  - grobgranulares Konzept auf höherer Abstraktionsstufe
  - näher am Geschäftsprozess-Modell
  - in dieser Vorlesung nicht besprochen

## 3.8. Vergleich Modelle auf Quellcode-Ebene

Komponentenmodelle auf Quellcode-Ebene:  
Aufbau von Anwendungen aus Software-Bausteinen

Ziel: Sicherung plattform- und sprachübergreifender  
Kompilierungskompatibilität

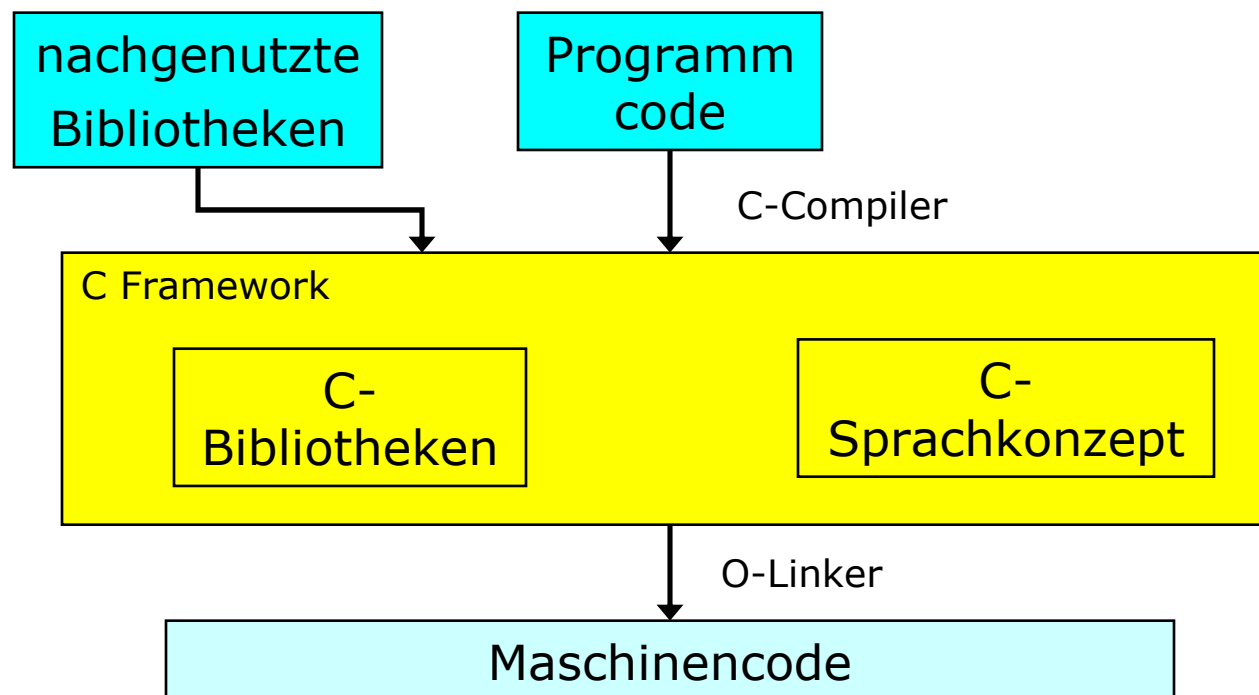
Anwendungsbereich: Desktop, Basiskomponenten

Grundlage: Gemeinsame Designprinzipien

Beispiele: C, Java, .NET

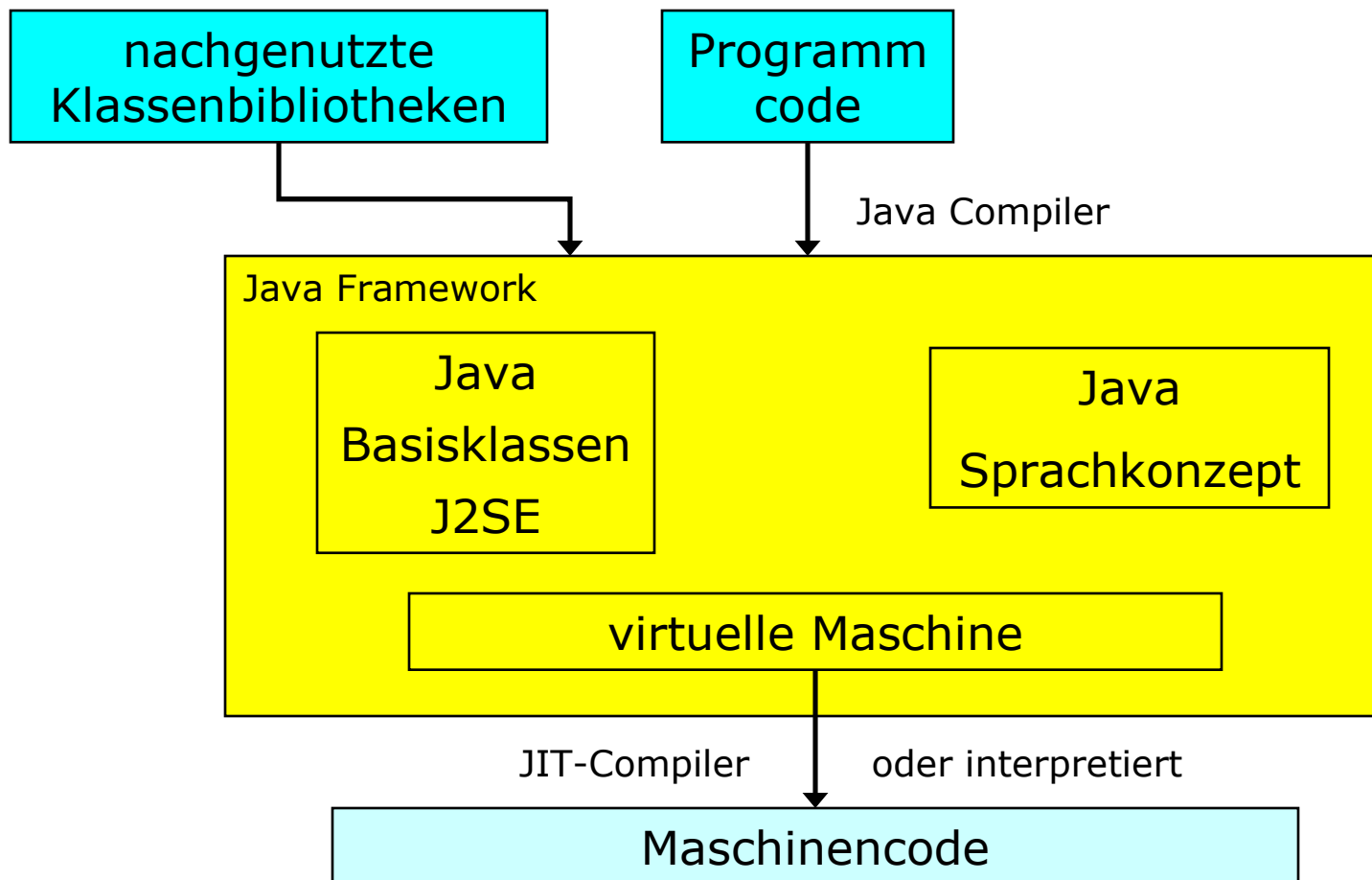
## 3.8. Vergleich Modelle auf Quellcode-Ebene

Eine Sprache, eine Plattform: C



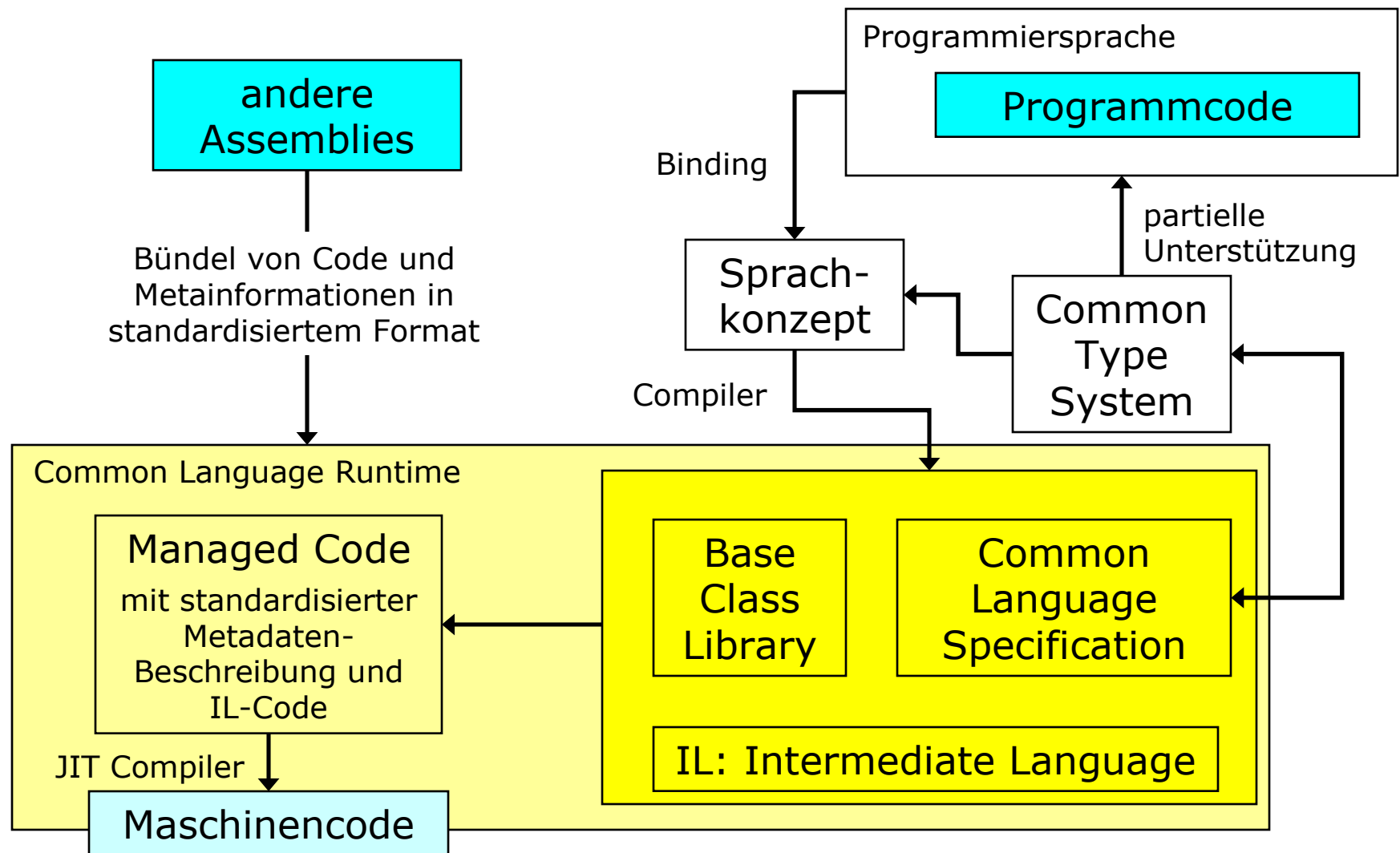
## 3.8. Vergleich Modelle auf Quellcode-Ebene

Eine Sprache, mehrere Plattformen: Java



# 3.8. Vergleich Modelle auf Quellcode-Ebene

Mehrere Sprachen, mehrere Plattformen: .NET



## 3.8. Vergleich Modelle für verteilte Anwendungen

Komponentenmodelle für verteilte Anwendungen:  
Aufbau von Anwendungen aus Komponenten

Ziel: Integration von Diensten in eine standardisierte verteilte Infrastruktur

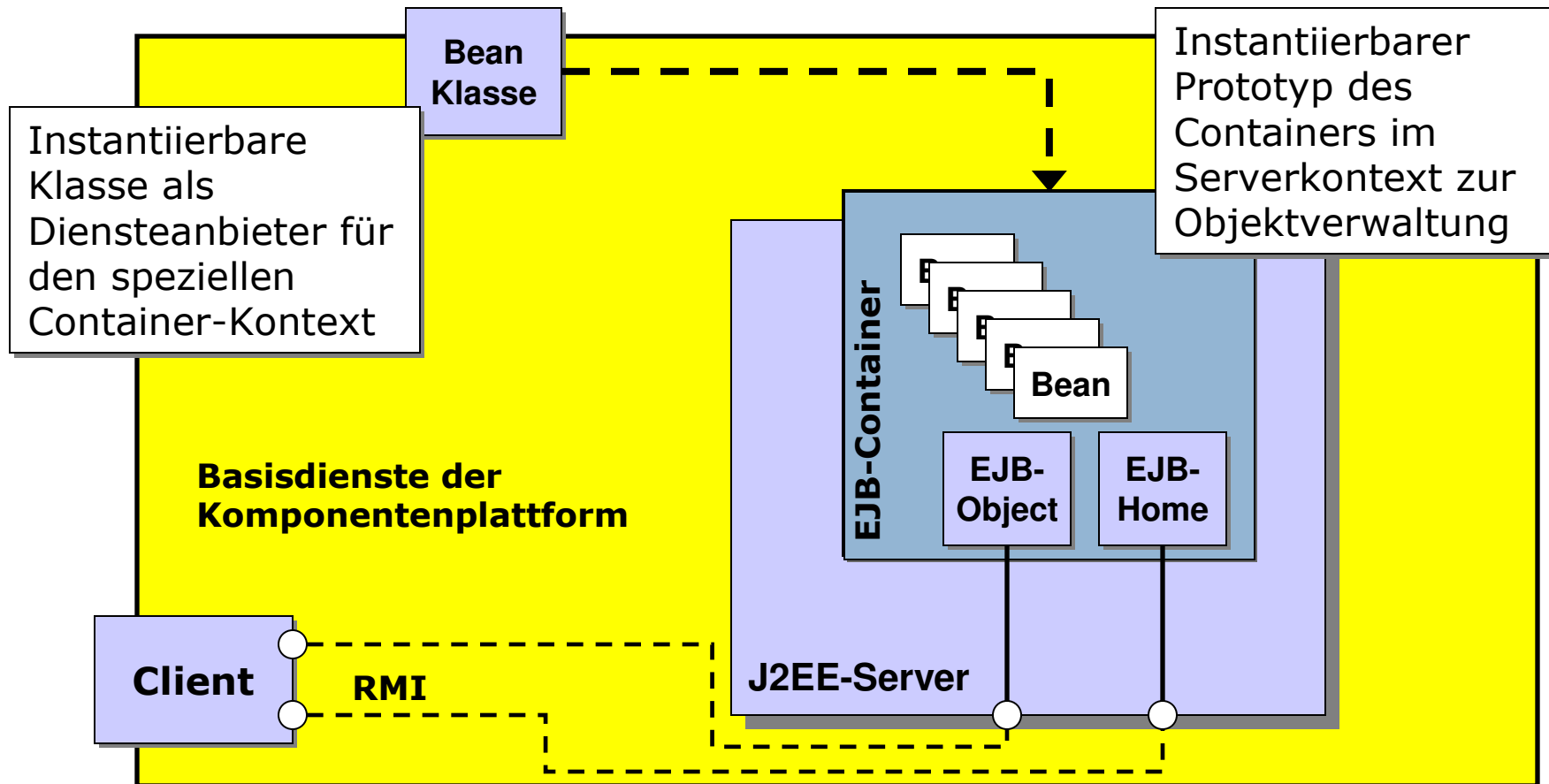
Anwendungsbereich: Middleware und verteilte Systeme

Grundlage: eng gekoppelte Client-Server-Architektur, gemeinsames Framework

Beispiele: EJB, Servlets, Server Pages, DCOM,

# 3.8. Vergleich Modelle für verteilte Anwendungen

## Prototypischer Aufbau am Beispiel EJB





### Konvergenz auf der Ebene der Konzepte

- Alle Zugänge unterstützen spätes Binden, Kapselung, dynamische Polymorphie, Vererbung auf Schnittstellenebene
- Standardisierte Komponenten-Transfer-Formats
  - Java: \*.jar, COM: \*.cab, CLI: Assemblies
- Uniformer Datentransfer
  - einheitliche Konzepte der Serialisierung von Objekten
  - Entwicklung von Persistenzmechanismen auf dieser Basis
- Ereignis- und Ereigniskanal-Konzept
- Metainformationen über Introspektion und Reflektion
  - erlaubt dynamische Erweiterung von Schnittstellen
- Einsatz von Konfigurationsinformationen
  - attributbasiertes Programmieren (CLI) – custom attributes
  - Montage-Beschreibungen
- dynamische Konfiguration
  - COM: QueryInterface, CORBA: EquivalenceInterface

Bestehende Differenzen können oft durch Brückenlösungen überwunden werden, die Drittanbieter zur Verfügung stellen.

Our motto is incompatibility is business – it's a huge opportunity for us.  
(A. O'Toole, CTO von IONA)

### Binärstandards

- COM: grundlegendes Ziel, wenn auch weitgehend ohne Bedeutung außerhalb der Windows-Welt
- Java: Binärstandard an JVM über JNI gebunden
- CORBA: Nur im Rahmen von CORBA-to-\* Compilern. Nicht plattformübergreifend standardisiert
- CLR: Ähnlich Java eine Ebene über Binärstandards angesiedelt, aber direkte und JIT-Compilation vorgesehen
  - CLR übersetzt entweder zur Installations- oder zur Ladezeit und führt immer nativen Code aus.

### Quellcodestandards für Kompatibilität und Portabilität

- CORBA: spezielle Sprachbindungen IDL-to-\*
  - Problem: Verwendung ORB-spezifischer Funktionen auf der Serverseite ist weit verbreitet
- Java: So lange alles in Java geschrieben ist – kein Problem
  - direkte Übersetzung aus anderen Sprachen in Java Bytecode möglich
  - Beispiel: J2EE-Implementierungen
- COM: keine Standards jenseits der Microsoft de-facto Standards
- CLR und .NET: Interoperabilitätskonzept durch Sprachbindungsstandards

### Speicherverwaltung und Garbage Collection

- Komplizierte Aufgabe in Systemen mit verteilten Objekten
- explizites Management des Lebenszyklus: CORBA
- Referenzzähler-Konzept: COM/DCOM
  - verlangt Kooperation aller Komponenten
  - skaliert schlecht in offenen verteilten Umgebungen
- Object leasing = Objektreferenzen haben nur beschränkte Lebensdauer
  - Java: GC von Java-RMI mit sehr guter Performance in verteilter Umgebung.
  - CLR: verwendet ähnlichen Ansatz

### Containerverwaltetes Persistenz-Management

- Mit EJB eingeführt und mit EJB 2.0 auch auf Relationen ausgeweitet
  - sehr datenbankspezifisch, außerhalb dieses Einsatzgebiets nicht sehr performant
- OLE-Datenbanken: Konzept der Persistenz-Abbildung (pluggable persistence mapping) erlaubt Abbildung auf verschiedene externe Speichermedien

#### Evolution und Versionsmanagement

- Sehr wichtig, wenn man Software-Entwicklung als Prozess verstehen will. Wird aber bisher kaum unterstützt
- COM: Schnittstellen und deren Spezifikation dürfen nach Veröffentlichung nicht verändert werden (immutable)
  - Aber: Möglichkeit der dynamischen Erweiterung
- CORBA: Versionsnummern, die zur Objektinitialisierung geprüft werden
  - Aber: dynamische Versionsprüfung nicht möglich
- Java: einige Regeln, aber inkonsistent
  - Problem der vorübersetzten Konstanten bei Versionswechsel
- CLI: Adressiert das Problem erstmals in voller Komplexität
  - Jede Assembly trägt Versionsinformationen von sich und allen Import-Komponenten. Es kann festgelegt werden, welche Toleranzen der Versionen erlaubt sind.
  - In einer Komponente können mehrere Versionen koexistieren
  - Standard wird weder von .NET noch von der CLR voll unterstützt

#### Kategorien

- erstmals von COM zur Klassifizierung von Software eingeführt
- Kategorie = Schnittstellenkontrakt auf Komponentenebene
  - Konkrete Komponente kann zu mehreren Kategorien gehören
  - Kategorie = abstrakte Zusicherung (high level assertion)
- Java, CORBA: kennen dieses Konzept nicht (aber: Marker-Interface)
- CLI: Unterstützung über Nutzerattribute

#### Montage / Konfigurierung

- Konzept der attributgesteuerten Programmierung
  - EJB: Attribute werden in der Montage-Beschreibung verwaltet
    - Faktorisierung des Montage-Schritts
  - J2EE: erweitert dieses Konzept auf andere Komponentenmodelle
  - CLR: kennt sowohl XML-basierte Konfiguration als auch CLI-basierte custom attributes
    - damit werden die Rollen des Komponentenentwicklers und des Komponentenmonteurs klarer unterschieden

### Komponenten und Objekte

Objektorientierung: Ist es das Evangelium und Synonym für Qualität schlechthin oder nur ein Weg, um Qualität zu erreichen?

Prinzipien:

- Alles wird in Objekte zerlegt, die Zustand und Verhalten kapseln.
- Objekte sind Instanzen von Klassen; letztere durch das (traditionelle) Vererbungskonzept verbunden.
- Objekte in polymorphen Kontexten

Vorbemerkung: Java = OO + Sprache, COM, CORBA, CLR sprachneutral

### Java

- Alles ist aus Objekten (Ausnahme: ein paar primitive Typen)
- Vererbung auf Schnittstellen- und Implementierungsebene
- Polymorphie durch Subklassen und Subschnittstellen
- Klassen (nicht Objekte!) als Einheit der Kapselung
  - orthogonal dazu das Konzept der Pakete
- RMI: Ortstransparenz von Objekten in verteilten Umgebungen
- Persistenz von Objektidentitäten auf der Ebene von Basisdiensten, nicht per se.

### COM

- Objekte nur über Schnittstellenmengen zugänglich
- keine zugänglichen Objektreferenzen, nur Schnittstellenreferenzen
- Objekte als Klasseninstanzen, aber ohne Vererbung
- Polymorphie wird durch n:m-Beziehung zwischen Schnittstellen und Klassen erreicht.
- Persistenz von Objektidentitäten auf der Ebene von Diensten



### CORBA

- Objekt als zentrales Konzept
- Klasse = Objektimplementierung, hat aber nichts mit Vererbung zu tun
- Mehrfachvererbung auf Schnittstellenebene, was ebenfalls die Basis für Polymorphie ist
- Kapselung durch Restriktion aller Interaktion auf Objektschnittstellen
- CORBA-Objekte sind recht gewichtig
  - Unterschied zwischen lokalen Objektreferenzen (kennt nur POA) und CORBA-Objektreferenzen
  - keine Unterstützung „kleiner“ oder „serverloser“ Objekte
  - zu teuer für jegliche Kommunikation innerhalb einer Komponente
  - OMG IDL kennt nur CORBA-Referenzen

### CLR

- Einheitliches Typsystem mit **Object** als Wurzel, das Wert- und Referenztypen vereint
  - Instanzen der Basistypen sind keine Objekte, können aber wie solche behandelt werden.

### CLR (Fortsetzung)

- Objekte als Klasseninstanzen
- einfache Implementations- und mehrfache Schnittstellenvererbung
- Persistenz von Objektidentitäten auf der Ebene von Diensten

### Zusammenfassung:

- Java und CLR kommen OO-Prinzipien am nächsten
  - Ausnahme: Klassen, nicht Objekte als Kapselungseinheit
- COM und CORBA: Kapselungseinheit Objektserver, aber keine Konzepte der Interaktion von Objekten im selben Server
- Java, COM, CLR: Unterscheiden zwischen internen und fernen Objektreferenzen. Letztere können nur über spezielle Infrastruktur (Java RMI, DCOM, CLR Remote) angesprochen werden

## Kontraktsspezifikation für Komponenten

- „Bessere Kontrakte für bessere Komponenten“ [Szyperski, 19.5]
- Anforderungen an die Kontraktsspezifikation sind höher als bei klassischer Software aus folgenden Gründen:
  - technologische Aspekte sind komplexer
  - Qualitäts-, Haftungs- und Sicherheitsfragen bei der Nutzung von Komponenten „Dritter“
- Wird in heutigen Komponentenkonzepten so gut wie nicht angesprochen
- QS ist nur bei klarer Spezifikation überhaupt kommunizierbar
  - Schnittstellen-Listing mit informeller Beschreibung (etwa auf der Ebene von **javadoc**) von Funktionalität reicht dafür nicht aus.
  - Qualität wird heute meist de facto durch starke Anbieter gesichert; am besten auch gleich im Kontext von Anwendungen dieser Anbieter
    - Beispiel: OLE und Word, Excel, Internet Explorer

## 3.9. Komponenten im Einsatz

### Kontraktsspezifikation für Komponenten

- Problem der Behinderung der Entstehung einer Komponentenwelt durch Unterspezifikation
  - Bsp. CORBA: vom BOA zum POA
  - Bsp. J2EE: von EJB zu EJB 2.0
  - Erfahrung kommt erst im praktischen Einsatz konkurrierender Implementierungen desselben Standards
  - Es geht um Konvergenz der Interpretation des Standards
    - ausgewogene Balance von Enge und Freiheit
- Schnittstellenkontrakt von Komponenten ist immer mehr als die Spezifikation des „Zusammenschaltens“
  - wird immer informelle Elemente enthalten, da es (auch) ein sozialer Kontrakt zwischen Entwicklern und Nutzern von Komponenten ist
  - klarer Link zwischen Schnittstelle (als „Kontrakt-Instanz“) und Kontraktsspezifikation erforderlich
- Zu jeder Schnittstelle gehört eine solche Spezifikation
  - Verbindung von Schnittstellen-Syntax und Semantik (Bedeutung)
  - COM-Konzept der unveränderlichen Schnittstelle ist Reflex dieser Tatsache
  - COM-UID = „Link“ zu einer solchen Spezifikation

## 3.9. Komponenten im Einsatz

### Kontraktsspezifikation für Komponenten

- Konzept der Kategorien: Spezifikation nach dem Baukastenprinzip
  - Java: Marker-Schnittstellen, COM: Kategorien
  - CORBA: Repository ID's verbinden eindeutige ID mit OMG IDL Typen
  - CLR: Konzept der Assembly sowie Konfigurationsattribute (custom attributes) zur Fixierung von Metadaten-Informationen
- Das sind bisher aber alles rein deklarative Methoden
  - Muss weiter formalisiert und in den Komponenten-Lebenszyklus (Entwicklung, Test, Zertifizierung, Laufzeit-Monitoring, ...) integriert werden
  - Entwicklungsrichtungen:
    - ASML = Abstract State Machine Language
      - u.a. Werkzeuge zur automatischen Generierung von Testorakeln und -fällen
    - TTCN = Test and Test Control Notation Language des ETSI (European Telecommunications Standards Institute)

## Komponentensoftware und die Grundlagen der Softwaretechnik

- Komponentenansatz enthält eine Reihe neuer Herausforderungen für einen modularen Ansatz auch in der Software-Technik
  - Schlüsselproblem: Ansatz der unabhängigen Erweiterbarkeit
  - späte Integration von Komponenten unabhängiger Hersteller
    - Konflikte mit Integrationstestkonzepten der klassischen SWT
  - Erweiterbarkeit muss selbst „designed“ werden, sonst passt nichts
  - Problem der verschiedenen methodischen Ansätze im SWE für interagierende Komponenten
  - Top-down-Design (ausgehend von der Anforderungsanalyse) trifft mit ziemlicher Sicherheit nicht die verfügbaren Komponenten
  - Bottom-up-Design ausgehend von Basisfunktionalitäten der verfügbaren Komponenten trifft mit ziemlicher Sicherheit nicht die Anforderungen
- Hier ist noch vieles unausgereift und Komponenteneinsatz nur aus strategischen Überlegungen heraus zu rechtfertigen.

## Komponentenorientiertes Programmieren als Methodologie

- Wie OOP die Methodologie des Programmierens objektorientierter Lösungen ist, so ist COP die Methodologie des Programmierens von Komponenten
- Definition (Szyperski):
  - Komponenten-orientiertes Programmieren bedeutet Unterstützung von
    - Polymorphie (Substituierbarkeit)
    - modulares Kapseln (Verstecken von Information)
    - spätes Binden und Laden (unabhängige Auslieferbarkeit)
    - Sicherheit (Typ- und Modulsicherheit)
- Bisherige Methodologien erstrecken sich nur auf die Entwicklung einzelner Komponenten
- Neuere Entwicklungen: Die Catalysis-Methode
  - <http://www.catalysis.org>

## Komponenten-Montage

- Komponenten als Einheiten der Auslieferung durch Dritte und als Einheiten der Komposition
  - Ein Weg zur Komposition ist traditionelle Programmierung
  - Attraktivität von Komponenten nimmt zu, wenn einfachere Kompositions-Prinzipien verfügbar sind
    - visuelle Komposition in Grafik-Werkzeugen
    - zusammengesetzte Dokumente
    - Zusammenbinden durch Skripting
    - Zusammenbinden als Web Services
  - besonders attraktiv, wenn der Endnutzer diese Montage selbst vornehmen kann (IKEA-Prinzip)
- Alle diese vereinfachten Montage-Prinzipien setzen auf inhaltlicher Seite kontextuelle Kapselung und Komposition der Komponenten voraus



## Infrastruktur-Aufwand für Komponentenanbieter

- OMA: Jeder ORB-Anbieter muss seine Sprachanbindung zu allen unterstützten Sprachen herstellen
- COM: benötigt COM-Infrastruktur, die es im Wesentlichen nur für Windows gibt
- Java: Überall lauffähig, wo eine JVM läuft
  - ein Classfile-Compiler pro unterstützter Sprache ist erforderlich
  - es gibt solche Compiler für viele gebräuchliche Sprachen
  - JVM-Standard ist allerdings für die Verwendung mit Java optimiert
- CLI: verfolgt ähnliche Strategie wie Java, zielt aber auf eine breitere Unterstützung von anderen Sprachen
  - braucht so was wie die JVM auf allen unterstützten Plattformen
    - CLR als Implementierung auf .NET (Windows, Microsoft)
    - Open-Source-Projekte Mono und Open CLI Library Project
    - FreeBSD-Version von Corel und Microsoft

**Der deutliche Sieger in diesem Rennen ist Java** und CLI ist der Versuch, diese Erfahrungen mit denen der COM-Welt zu vereinigen

## 3.9. Komponenten im Einsatz

### Lessons learned

Folgerung: Komponentenkonzepte müssen in eine (technische) Infrastruktur eingebettet sein.

Eine Lehre aus CORBA:

Wenn zu viele Dimensionen von Freiheit gekoppelt werden, um eine möglichst große Variation von Lösungen zu ermöglichen, dann werden die meisten praktischen Lösungen nur für Marktnischen relevant sein.

CORBA versagt bei einem seiner zentralen Versprechen: eine breite Varietät nicht nur von möglichen, sondern von realen Lösungen zu unterstützen. Es fehlen dafür strenge low-level Integrationsstandards.

Die Maximierung der Zahl der kombinatorisch möglichen Variationen minimiert die Zahl der real verfügbaren Varianten.

Für ein Komponentenmarkt ist die Freiheit der Inhalte ebenso entscheidend wie die Beschränktheit der Design-Konzepte.

Diese Standardisierungsbemühungen stehen noch ganz am Anfang.

## Komponenten und Berufsprofile für Informatiker

### Komponenten-Systemarchitekt

- Komponenten funktionieren nur innerhalb eines Frameworks (konkrete Implementierung eines Architektur-Konzepts)
- Konsistentes Architekturkonzept deckt mehrere Frameworks und deren Interoperabilität ab
- Entwickelt die Architektur für die Architekten - der einzelnen Frameworks

### Komponenten-Frameworkarchitekt

- Entwicklung von Konzepten und Werkzeugen, um konkrete Komponenten in eine Infrastruktur „einzustöpseln“
- Muss sich im gesamten Anwendungsbereich des Frameworks gut auskennen
- Implementierung des Frameworks ist die Basis für eine funktionierende Komponentenwelt
- Muss Anforderungen an die Komponenten-Entwickler spezifizieren

#### Komponenten-Entwickler

- Komponenten-Entwickler erstellen die „Blätter“ für das Komponenten-Framework
- Die funktionalen Spezialisten in dieser arbeitsteiligen Struktur mit Spezialkenntnissen aus den konkreten Anwendungsbereichen, die von den zu entwickelnden Komponenten abgedeckt werden

#### Komponenten-Monteur

- Aufgabe: Anpassen, „Zuschneiden“ und Integrieren von Komponenten für den konkreten Gebrauch in speziellen Anwendersystemen
- Auflösung des klassischen Begriffs der „Anwendung“ als monolithisches und statisches System zugunsten des Konzepts einer organischen (und organisch wachsenden) IT-Infrastruktur
- End-Nutzer übernehmen in einem solchen Konzept zunehmend eine eigenständige Rolle, die vom Komponenten-Monteur abzugrenzen ist
- Aspekte der Nutzerschulung treten dann ergänzend hinzu
- Feedback zu Komponenten-Entwicklern und Framework-Architekten