

3. Vorgehensmodelle

- „Voraussetzung für den sinnvollen Einsatz von Notationen und Werkzeugen zur Softwareentwicklung ist ein Vorgehensmodell, das den Gesamtprozess der Softwareerstellung und -pflege in einzelne Schritte aufteilt und die Verantwortlichkeiten der beteiligten Personen (Rollen) klar regelt.“

Beispiele für Vorgehensmodelle

- Klassisches Wasserfallmodell
- Spiralmodell nach Böhm
- RUP (Rational Unified Process)
- XP (eXtreme Programming)
- V-Modell

3.1 Rational Unified Process

- Basiert auf Standard-OO-Modellierungssprache **UML** und dem zugehörigen Vorgehensmodell
- Bestandteile
 - Verschiedene Generationen
 - Generationen bestehen aus verschiedenen Phasen mit Iterationen
 - Inception (Vorbereitung)
 - Elaboration (Entwurf)
 - Construction (Konstruktion)
 - Transition (Einführung)
- Iterationen

3.1.2 Eigenschaften des RUP

- Modellbasiert
Modelle (Dokumente) für die einzelnen „Schritte“ des Prozesses
- Prozessorientiert
genau definierte Abfolge von wiederholbaren Aktivitäten
- Iterativ und inkrementell
- Risikobewusst
Aktivitäten mit hohem Risiko in frühen Iterationen
- Zyklisch
Ergebnis jedes Zyklus:
Neue **Systemgeneration** (als kommerzielles Produkt ausgeliefert)
- Ergebnisorientiert
Meilenstein (def. Ergebnis zu def. Zeitpunkt) steht am Ende jeder Iteration

- Faustregeln für die Ausgestaltung eines Entwicklungsprozesses
- Rollenbasierte Softwareentwicklung und Arbeitsbereiche
- Projekte funktionieren, wenn die richtigen Personen unter Verwendung geeigneter Sprachen, Methoden und Werkzeuge kooperieren
- Zuweisung von Rollen an Personen klärt Verantwortlichkeiten und Kompetenzen
- Vorteile:
 - Standardisierungsvorschlag und kommerzielles Produkt
 - Eigene Managersicht
 - Feinere aktivitätsorientierte Sicht für Entwickler
- Nachteile:
 - Komplexes, noch stark in Veränderung befindliches Vorgehensmodell

3.1.3 Phasen des RUP

3.1.3.1 Inception

- Etablieren des Geschäftsprozesses
 - Risikoanalyse, Erfolgskriterien, Ressourcenschätzung, Milestones
- Definition des Projektumfangs
- Finden der Akteure und Anwendungsfälle
- Spezifikation der wichtigsten Anwendungsfälle
- Eventuell Prototyp

- Meilenstein:
 - Ziele für den geplanten Lebenszyklus
 - Entscheid über die Weiterentwicklung

RUP Lifecycle



3.1.3.2 Elaboration

- Analysieren der Problemstellung
- Etablieren einer Architektur
- Klärung der größten Risiken
- Finden der Akteure und Anwendungsfälle
- Beschreibung fast aller Anwendungsfälle
- Erstellen eines Prototypen mit
 - Zielarchitektur
 - Den wichtigsten Anwendungsfällen
- Meilensteine:
 - Detaillierte Untersuchung der Ziele und des Leistungsumfangs
 - Wahl der Architektur
 - Auflösung der Hauptrisiken

3.1.3.3 Construction

- Iterative und inkrementelle Fertigstellung der Software
 - Beschreibung der letzten Anwendungsfälle
 - Konkretisierung des Designs
 - Vervollständigung der Implementierung
 - Testen

- Meilenstein:
 - Entscheid ob Software, Installationsorte und Benutzer für die Ausbreitung bereit sind.

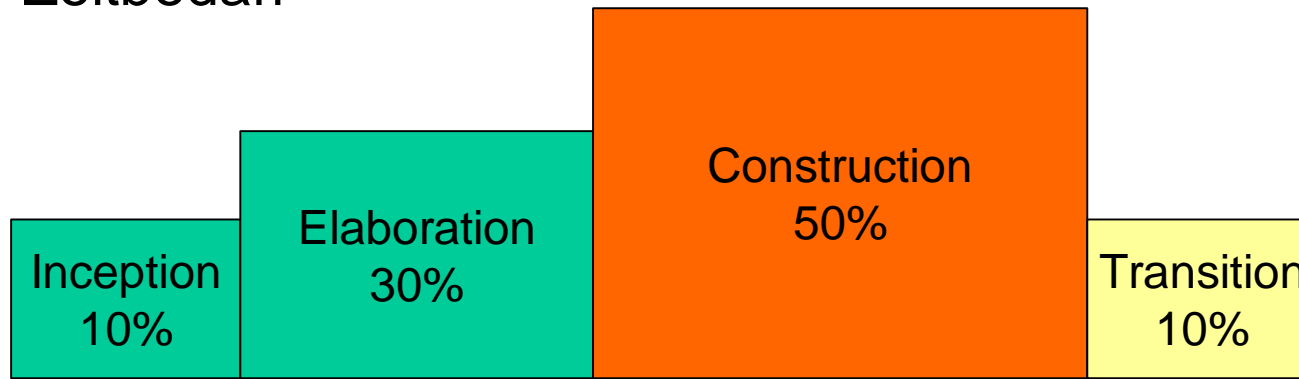
3.1.3.4 Transition

- Übergabe der Software an den Benutzer
- Daraus ergeben sich
 - Systemanpassungen
 - Fehlerkorrekturen
- Diese Phase beginnt mit der beta-Release

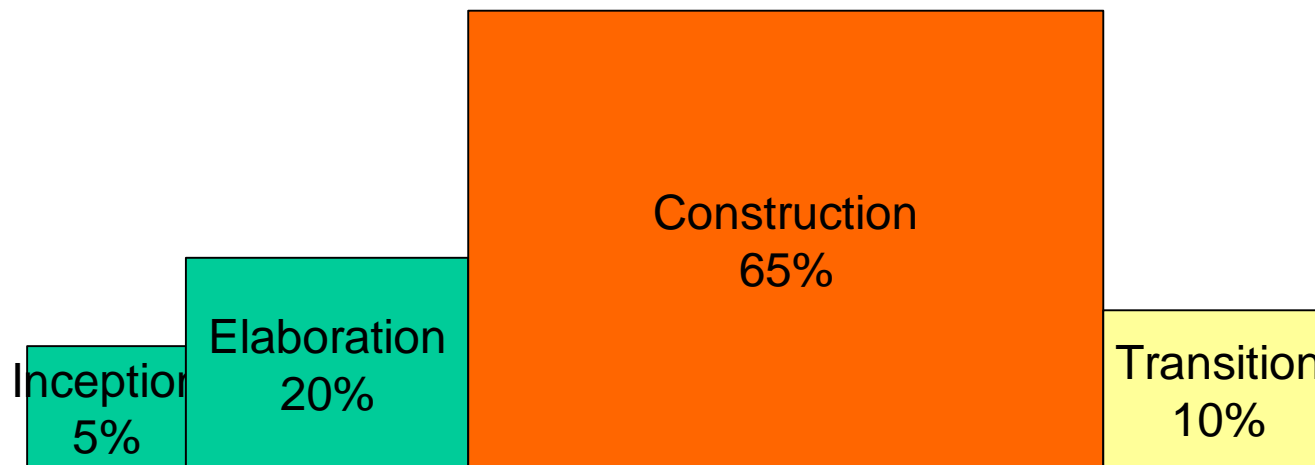
- Meilensteine:
 - Entscheid ob die Ziele für den Lebenszyklus erreicht wurden
 - Entscheid ob ein weiterer Zyklus gestartet wird
 - Feedback zum Prozess und Verbesserungsvorschläge

3.1.4 Zeitbedarf und Kosten für die einzelnen Phasen (Richtwerte)

- Zeitbedarf



- Kosten



3.1.5 Planung der Iterationen

- Die Zeitdauer der Iteration ist abhängig von der Anzahl der daran beteiligten Mitarbeiter

| Anzahl Personen | Dauer der Iteration |
|-----------------|---------------------|
| 5 | 1-2 Wochen |
| 15 | 1 Monat |
| 45 | 6 Monate |
| 100 | 12 Monate |

3.2 eXtreme Programming

3.2.1 Prinzipien des XP

- Einfachheit
 - Do the simplest thing you can do
 - Nur inkrementelle, lokale Änderungen (refactoring)
 - Alles nur einmal tun (extreme reuse)
- Kommunikation
- Programmieren in 2er Teams
- Feedback
 - Dauernder direkter Kontakt zum Anwender
 - Kontinuierlich neue Versionen der Software
- Courage
 - Extensive unit tests

3.2.2 Vergleich zu RUP

- So wenig Prozess wie möglich, so viel wie nötig
- Für kleine und mittlere Teams
- Kommunikation ersetzt Großteil der Dokumentation

3.2.3 XP Praktiken

3.2.3.1 Das Planspiel

- Ziel: Planung von Umfang, Zeit und Kosten des nächsten Release
- Vorgehen:
 - Schreiben der *Stories*
 - Schätzen von Aufwand und Risiko
 - Auswahl der Stories
 - Ständige Korrektur des Plans

3.2.3.2 Einfaches Design

- Da die Zukunft unsicher ist, wird nur das implementiert, was jetzt gebraucht wird. Nur dafür zahlt der Kunde.
- Kriterien für ein einfaches Design:
 - Es besteht alle Tests.
 - Es enthält keine Redundanzen.
 - Es legt offen, was die Programmierer intendieren.
 - Es hat die geringste mögliche Anzahl von Klassen und Methoden.

3.2.3.3 Testen

- Programmierer schreiben *unit tests*
- Kunden spezifizieren Anwendungstests

3.2.3.4 Refactoring

- Ständige Restrukturierung des Codes
 - Zur Vermeidung duplizierter Logik
 - Zum Erhalt des einfachsten Designs
 - Zur Verbesserung der Kommunikationsfähigkeit des Codes
- Erst refaktorisieren, dann neue Features implementieren, dann wieder refaktorisieren.
- Unit Tests sind das Fangnetz der Refaktorisierung.

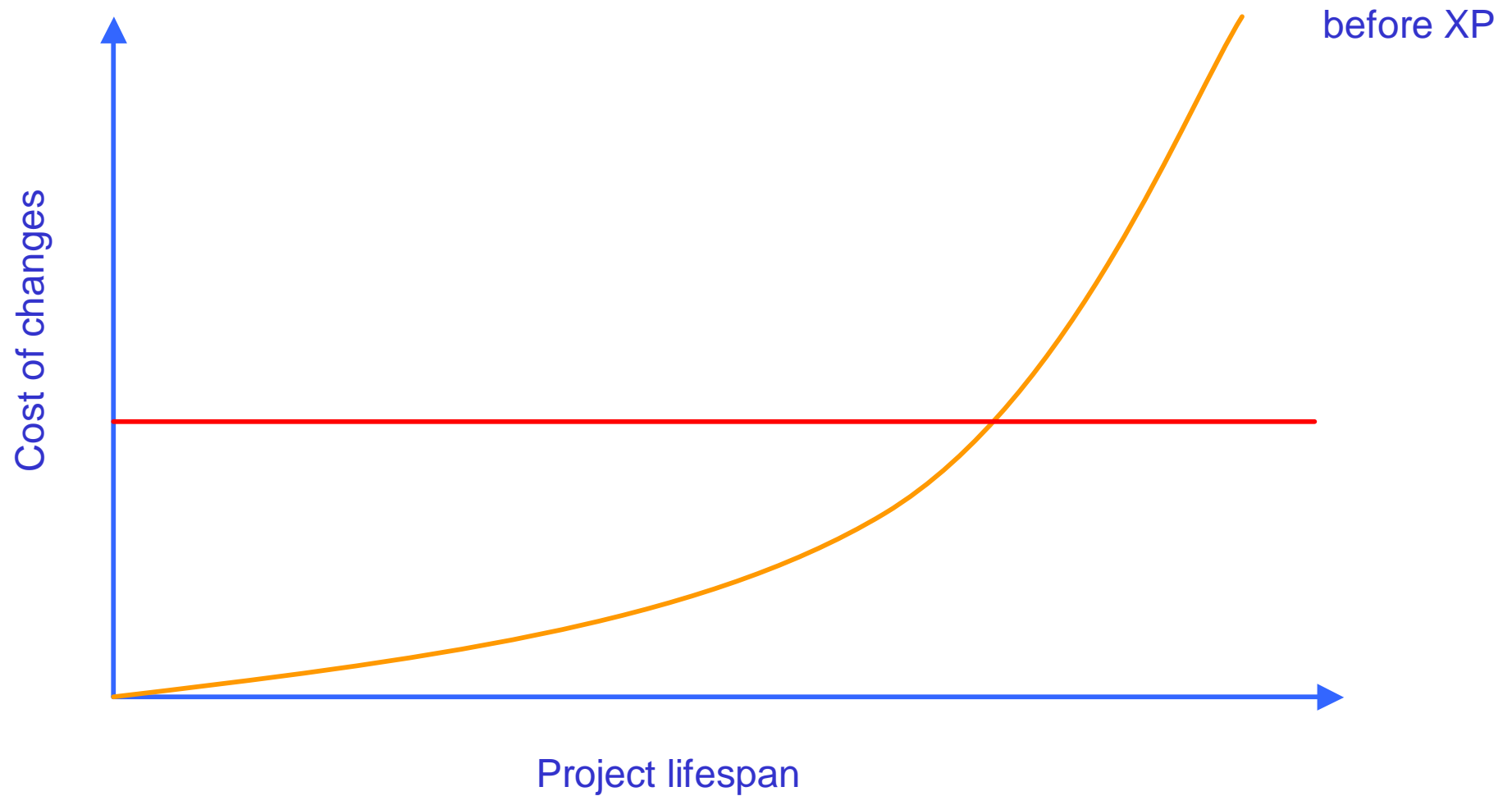
3.2.3.5 Pair Programming

- Test und Code werden grundsätzlich zu zweit entwickelt.
- Verteilt das Wissen auf das ganze Team.
- Sorgt für die Einhaltung der Programmierstandards und der Unit Tests.

3.2.3.6 Der Rest

- Kunde im Entwicklungsteam
 - Ständiger Kontakt zum echten Einsatz beim Kunden
- Kleine Releases
 - Nur kleine Änderungen, mit wenigen Funktionen starten, frühzeitige Releases
- Ständige Integration
 - Tägliches Einbinden der Änderungen nach unit tests
- Einfache Metapher leitet die Entwicklung
 - Stellt Namens-Konventionen zur Verfügung
- Kollektiver Codebesitz
- Programmierstandards
 - Jeder coded den selben Standard
- Max. 40-Stunden-Woche

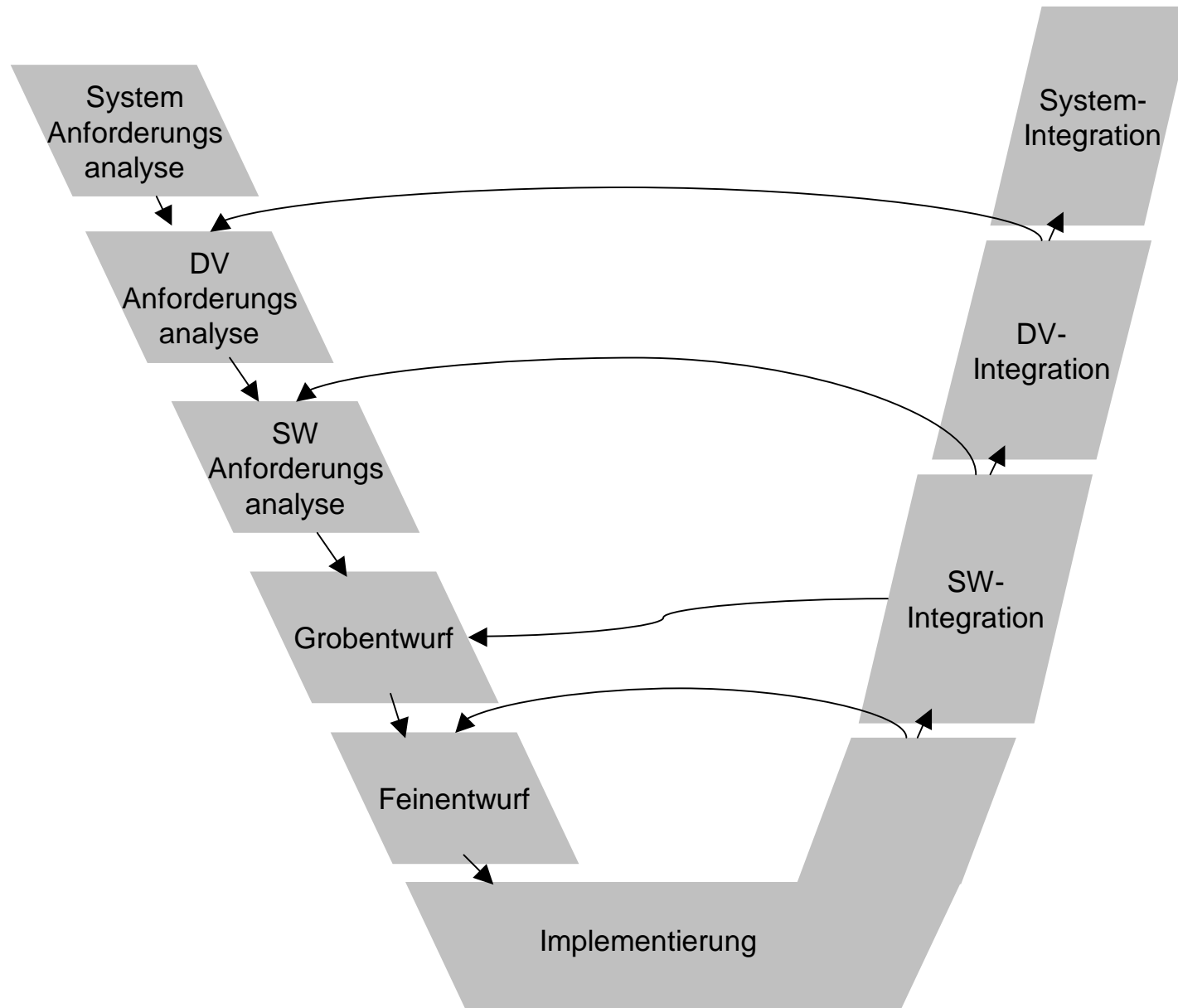
3.2.4 Vorteile durch XP



3.2.5 Mapping XP zu RUP

| XP | RUP |
|---|--------------------------------------|
| Stories Additional documentation from conversations | Vision Glossary Use-Case Model |
| Constraints | Supplementary Specifications |
| Acceptance Tests Unit Tests Test Data Test Results | Test Model |
| Software-Code | Implementation Model |
| Releases | Product Release Notes |
| Metaphor | Software Architecture Document |
| Coding Standards | Design Model |
| Release Plan | Software Development Plan |
| Overall plan | Business Case Risk List |

3.3 V-Modell (1)



V-Modell (2)

- Ursprung im Wasserfallmodell
- Linke Schenkel beinhaltet Projektentwurf bis zur Implementierung
- Rechter Schenkel beinhaltet Integration und Tests der gegenüberliegenden Entwicklungsphasen
- Für öffentliche Aufträge bindend
- Legt fest
 - Welche Aktivitäten in den einzelnen Phasen durchzuführen sind
 - Wer die Verantwortung hat
 - Welche Werkzeuge verwendet werden
- Verantwortlichkeiten in Form von Rollen

4. Komponenten

Komponente

Def: Ein Softwaremodul mit ausführbaren Bestandteilen, eigener Identität und wohldefinierten Schnittstellen.

Wozu?

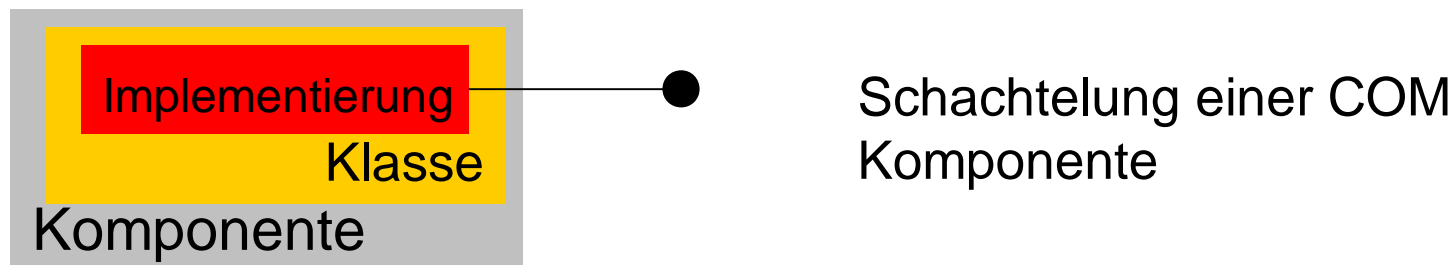
- Ähnliche Motivation die zu OOSE führte
 - Software-Krise
 - Wiederverwendung
 - Qualitätsverbesserung
 - Kürzere Entwicklungszeiten (Time-to-market)
 - Black-box reuse anstatt white-box reuse
 - Entwickler gibt kein Know-How Preis
 - Plattform- und Sprachenunabhängigkeit

4.1 Komponenten Modelle bzw. Architekturen

- Microsofts Component Object Model
- Common Object Request Broker Architecture CORBA
- JAVA Beans
- ...

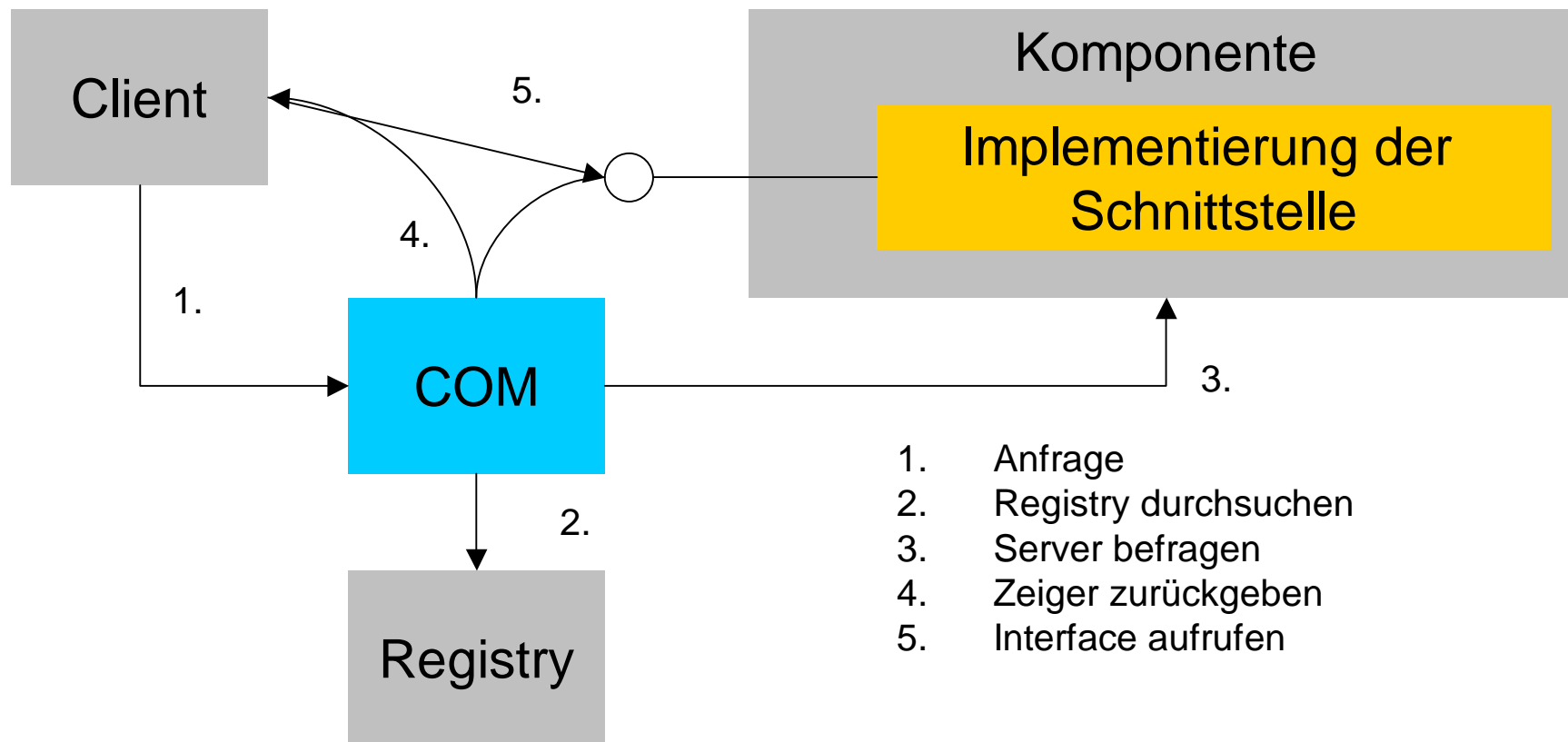
4.2 Microsofts Component Object Model (COM)

- COM definiert einen Standard für die Kommunikation von Objekten unter Windows.
- Clients interagieren mit COM-Objekten über einen Server, der als ausführbare Datei oder als dynamische Bibliothek vorliegen muss.
- COM als das „bessere“ C++.
- Trennung von Schnittstelle und Implementierung.
- Ausweg aus der .dll-Hölle.
- Bessere Software-Verteilung und Portabilität.
- IDL für die Definition der Schnittstellen.



Microsofts Component Object Model (COM) (2)

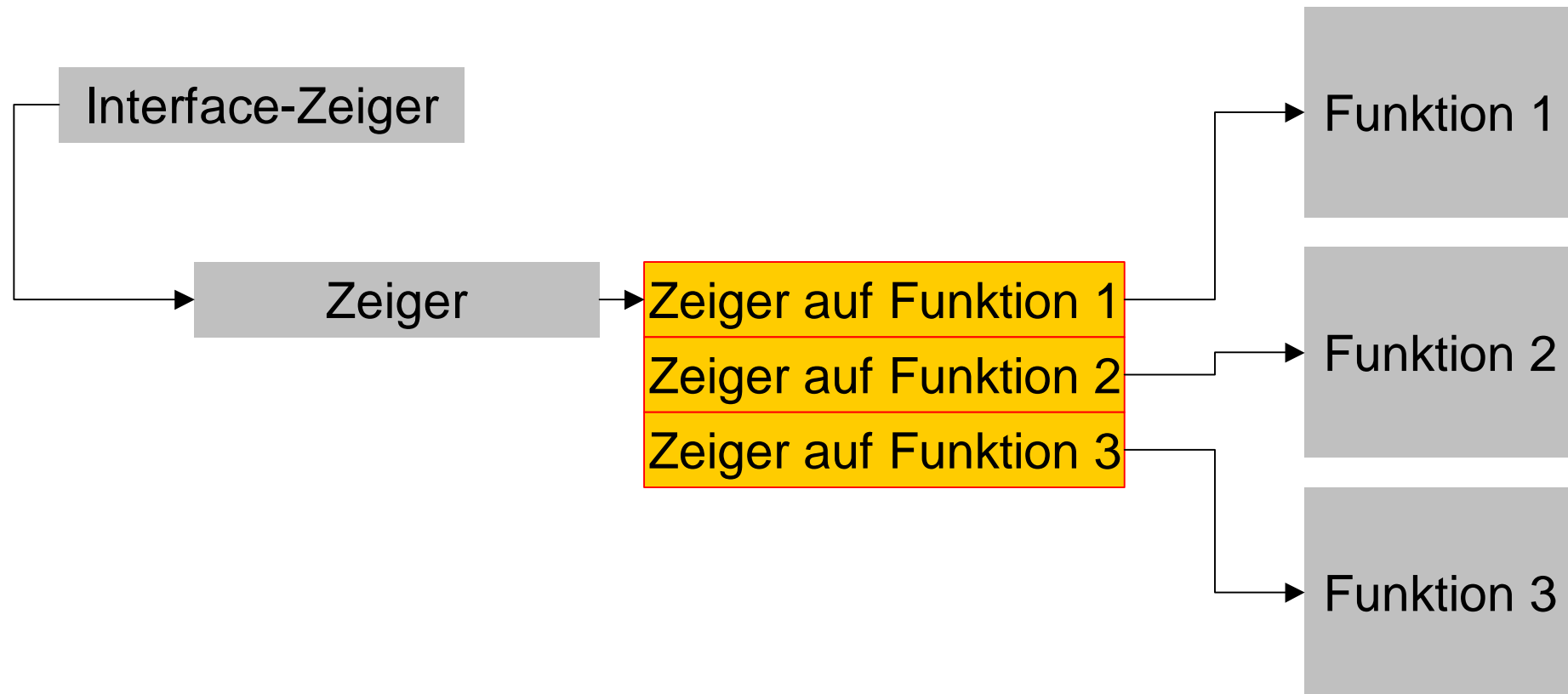
Trennung von Interface und Implementierung



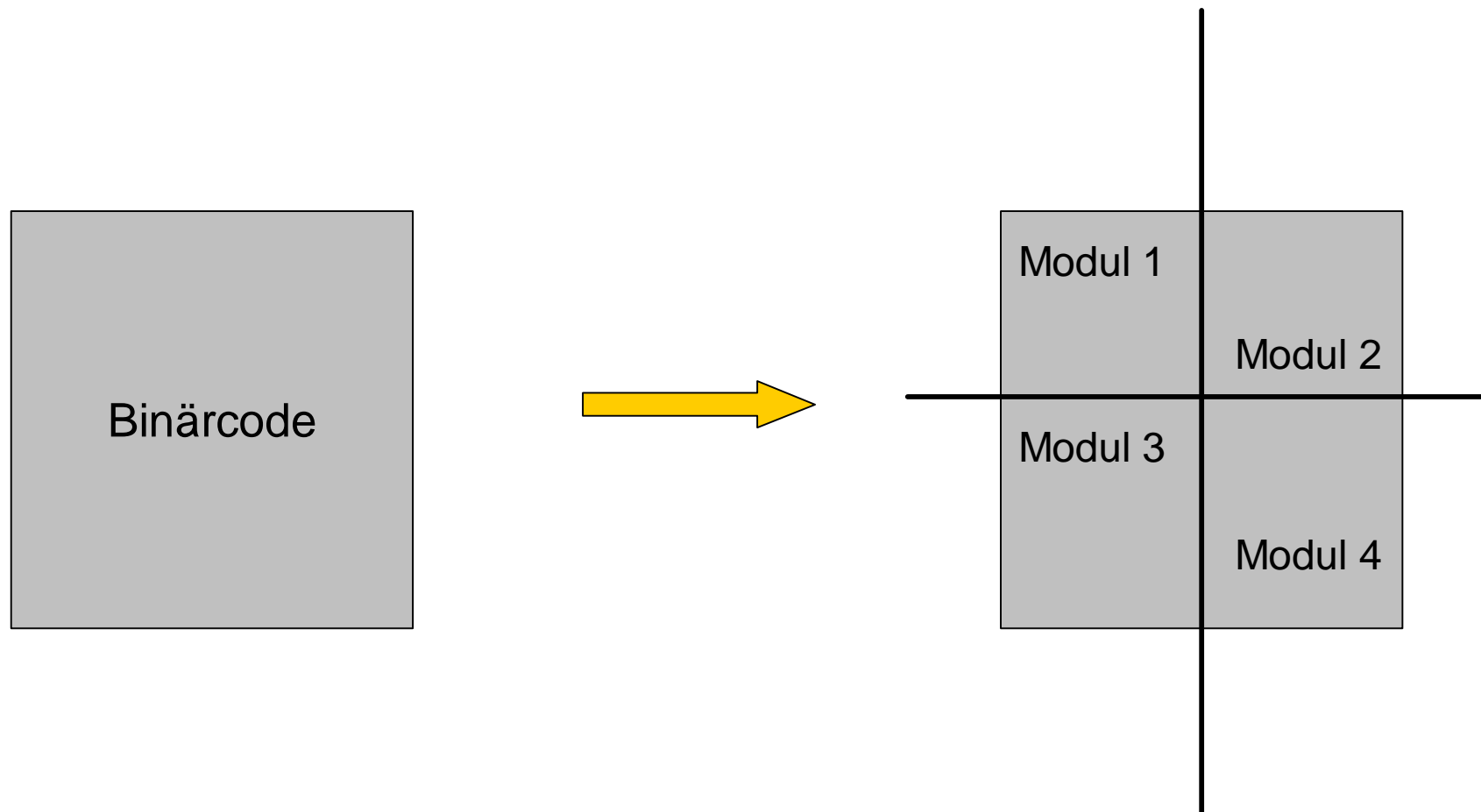
1. Anfrage
2. Registry durchsuchen
3. Server befragen
4. Zeiger zurückgeben
5. Interface aufrufen

Microsofts Component Object Model (COM) (3)

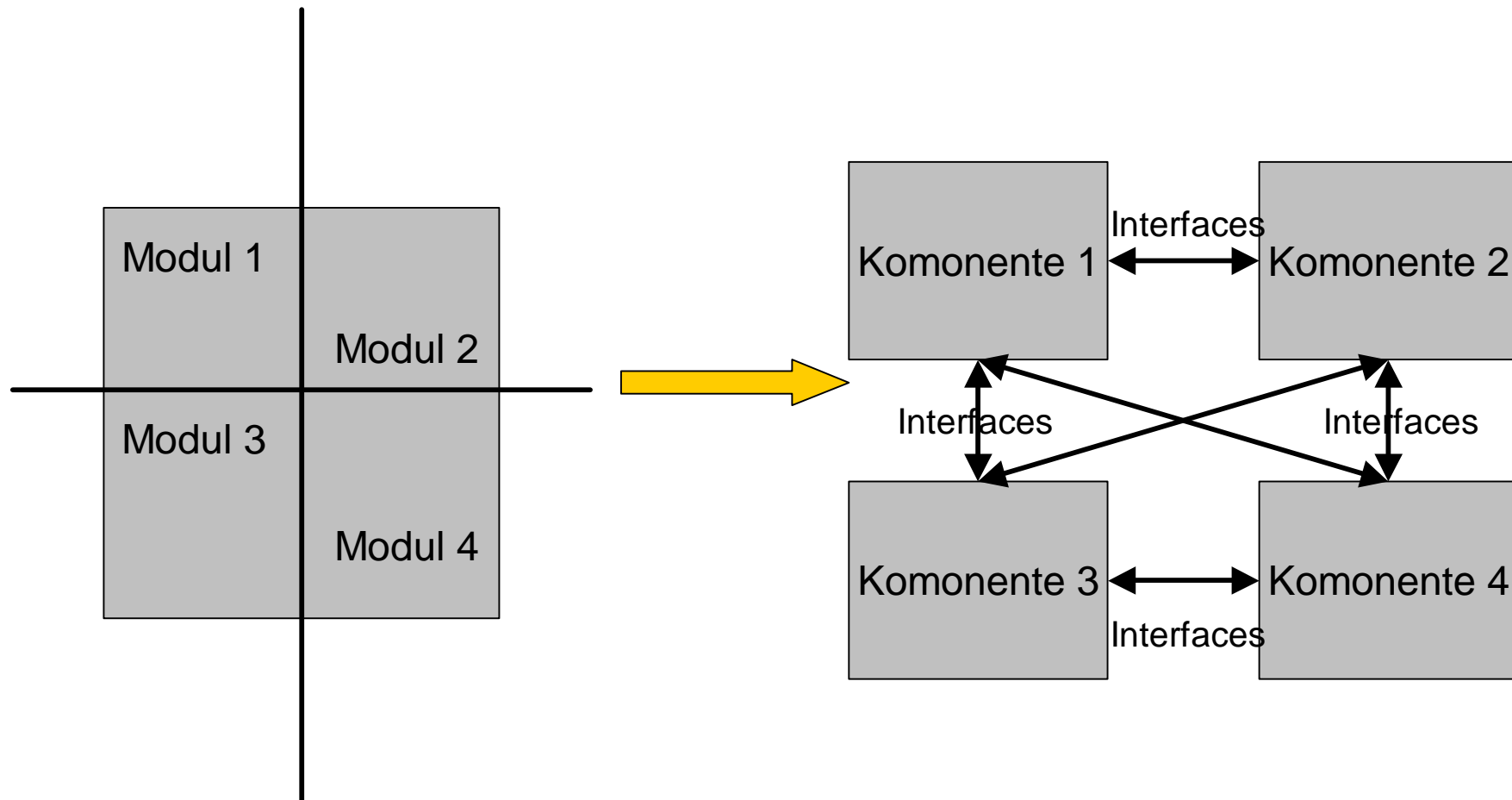
- Zugriff auf Objekte über Tabellen



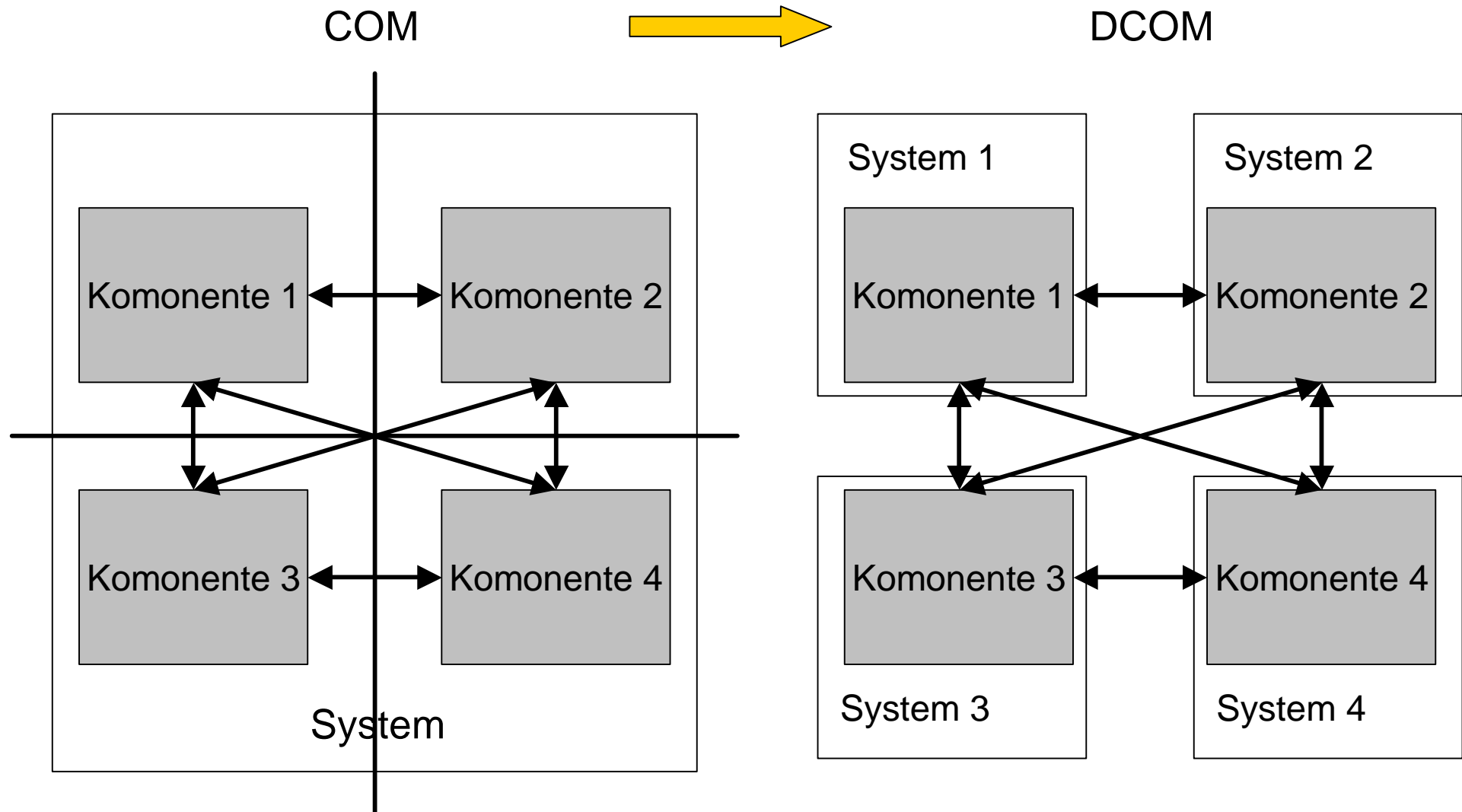
4.3 Von monolithischen Systemen über COM zu .net



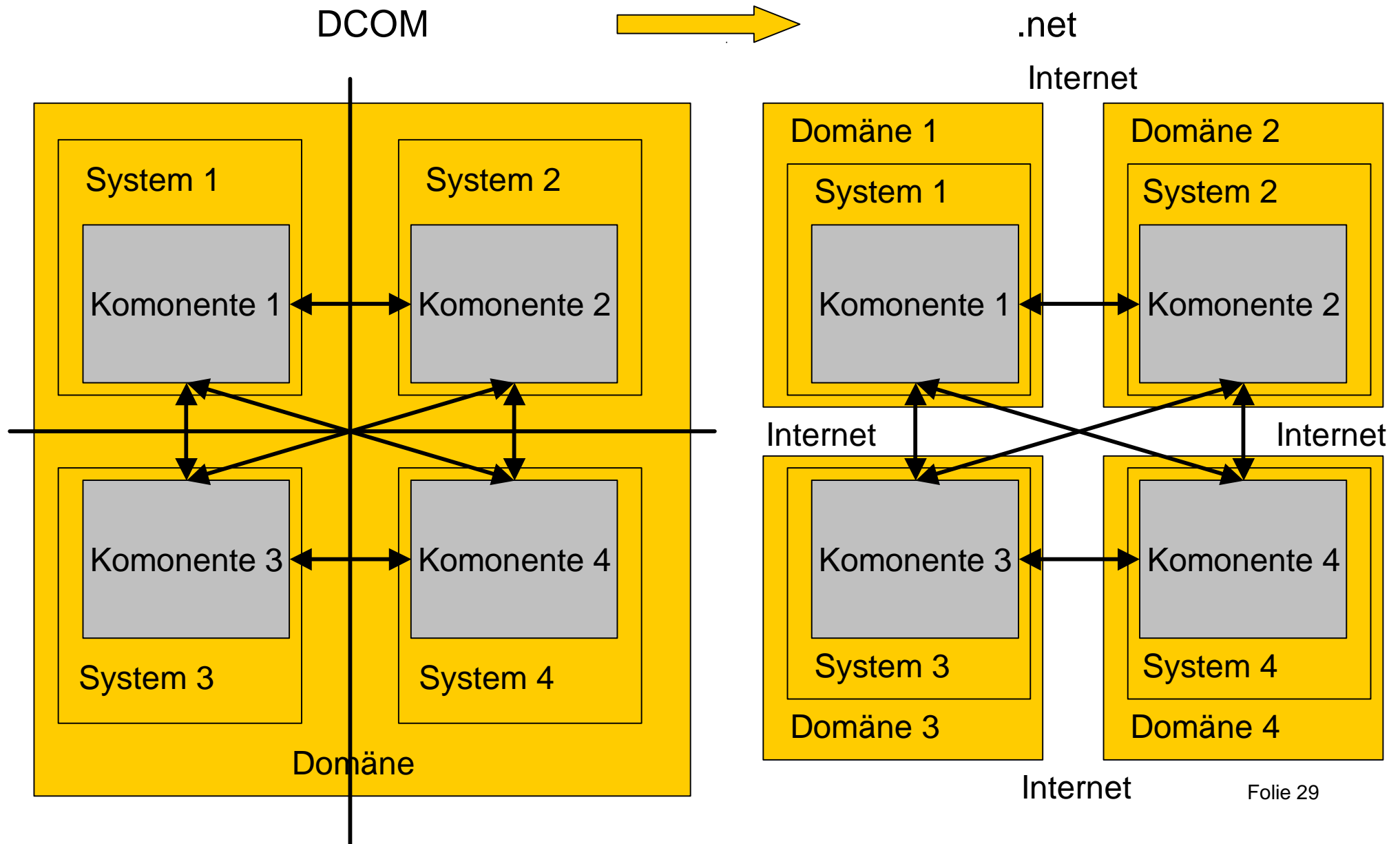
Von monolithischen Systemen über COM zu .net (2)



Von monolithischen Systemen über COM zu .net (2)



Von monolithischen Systemen über COM zu .net (2)

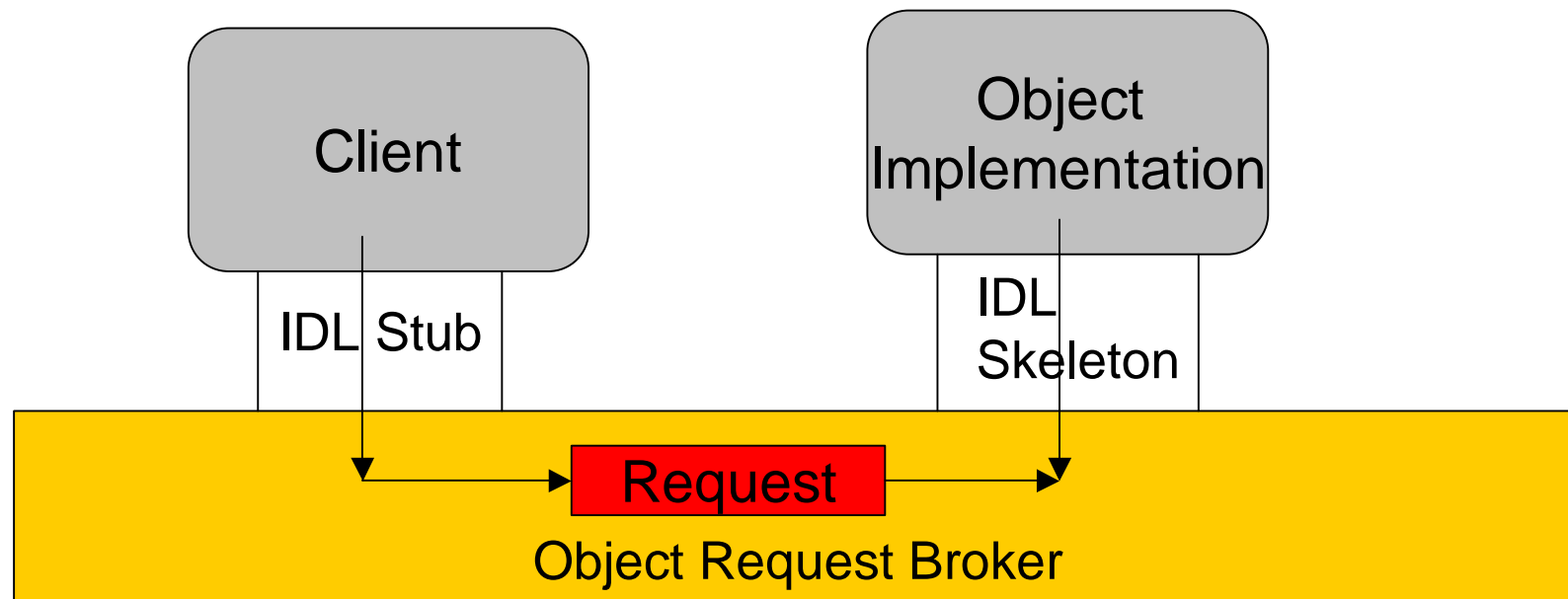


4.4 CORBA

- CORBA steht für **C**ommon **O**bject **R**equest **B**roker **A**rchitecture
- Erste Version 1991 von der Object Management Group (OMG) definiert.
- Ziel: Middleware zu schaffen, welche eine orts-, plattform- und implementationsunabhängige Kommunikation zwischen Applikationen erlaubt.
- ORBs (*Object Request Broker*) sind die technischen Implementationen des Standards CORBA
- Ein ORB ermöglicht es einem Client, eine Meldung transparent an ein Serverobjekt zu senden, wobei das Serverobjekt auf derselben oder einer anderen Maschine laufen kann
- Seit 1994 Meldungs austausch zwischen ORBs unterschiedlicher Hersteller möglich.

CORBA (2)

- Der ORB ist dafür zuständig, das Serverobjekt zu finden, dort die Funktion aufzurufen, die Parameter zu übergeben und das Resultat an den Client zurückzureichen.



CORBA (3)

- Trennung zwischen Schnittstelle und Implementierung
- Definition der Schnittstellen über IDL (Interface Definition Language)
 - IDL ist eine implementations-unabhängige Beschreibungssprache
- In einem zweiten Schritt erst wird diese Definition ausprogrammiert
 - Sprachenunabhängig, z.B.: Client in Java und Server in C++
- Services ergänzen die Kernarchitektur
 - Naming-Service: hilft bei der Suche und Identifikation von Objekten
 - Event-Service: definiert die Schnittstellen für Messaging-Systeme
 - Transaktions-Service: Schnittstellen für Two-Phase-Commit und andere Verfahren
 - Datenbank-Services: einheitliche Schnittstellen für Datenbankoperationen

4.5 JAVA Beans

Spec.: „A Java Bean is a reusable software component that can be manipulated visually in a builder tool“

- Einfache Programmierschnittstelle
- Plattformneutrales Komponentenmodell
- Verbindungen zu anderen Komponenten- und Verteilungsmodellen wie DCOM oder CORBA
- In plattformabhängigen Behältern (containern) einbettbar z. B.: MS Excel, Word etc.
- Beispiele: GUI widgets, Spreadsheet, HTML- Browser
- Die Granularität von JavaBeans reicht von einer Klasse bis zur ganzen Applikation.
- Eine oder mehrere Klassen realisieren eine Komponente
- Explizite Unterstützung von visueller Programmierung
- Visible und Invisible Beans

JAVA Beans (2)

- JAVA Bean = Java Objektmodell (Klasse) +
Eigenschaften (properties) +
Ereignisse (events) +
Untersützung für visuelle Komposition