# Modelle des verteilten objektorientierten Programmierens

**Nils Schmeisser**
**Forschungszentrum Rossendorf e.V.**

# Inhalt

- Objektorientierter Entwurf
- Formales Objektmodell (nach Abadi/Cardelli)
- das Objektmodell der Object Management Group (OMG)
  - die Common Object Request Broker Architecture (CORBA)
- das Microsoft Component Object Model (COM)
  - die Erweiterung auf Distributed COM
- das Java Objektmodell
  - Java Remote Method Invocation (RMI)
  - Java und CORBA Bindung
- Vergleich
- Diskussion

# OO Entwurf I

- objektorientiertes Herangehen an eine Problemloesung beinhaltet drei Teilaspekte
  - *Analyse:* die zu simulierenden realen Komponenten sollen ihr Analogon in der modellierten Welt wiederfinden
  - *Design:* die (Software) Modelle sollen so gebaut sein, dass sie moeglichst eindeutig auf das abstrakte Modell abgebildet werden koennen; der Fokus des Designs liegt auf der Systembeschreibung und nicht auf der Beschreibung von Algorithmen, Alg. werden in Methoden '*faktorisiert*'
  - *Wiederverwendung:* waehrend der Analyse- und Designphase entsteht eine Hierarchie, die die Wiederverwendung von Methoden erlaubt

# OO Entwurf II

- **Wiederverwendung**
  - **eine Softwarekomponente heisst 'wiederverwendbar', wenn sie auf einfache Art und Weise in mehr als einem Kontext genutzt werden kann**
    - **traditionelles prozedurales Herangehen erfordert hierfuer einen streng einzuhaltenden Kontrakt**
    - **in OO Sprachen/Systemen ist dies nicht noetig**
      - **ein Objekt kann durch jedes andere mit mindestens der gleichen Menge von Attributen ersetzt werden (*Polymorphie*)**
      - **dieses Verfahren erlaubt das *Ueberschreiben*, die *Verfeinerung* und die *Uebernahme* von Methoden; neue Objekte entstehen als Mischung aus beidem**

# Formales Objektmodell

- *Sorte:* **Komposition**
  $\omega = i1:t1 + ... + in:tn$
- *Abstrakter Datentyp (ADT):*
  **Algebra aus einer Menge $\Omega$ von Sorten, einer Menge darueber erklaerter Operatoren $\Delta$ und deren Spezifikation $\Gamma$ ($\Omega\Delta\Gamma$-Algebra)**

```
KREIS=     x:REAL+y:REAL
  +name:STRING

KREIS    is
  x:REAL;
  y:REAL;
  name:STRING;
end    KREIS;
```

- **nach Abadi/Cardelli**
  - **Tupel von 'pre-Methoden' + Konstruktor = Klasse**
  - **Tupel von gebundenen Methoden = Objekt**

# Objektmodell der OMG I

- abstraktes Modell (keine Vorgabe einer Implementation)
- *Objekt-System*
  - Client-Server-Architektur
- *OMG Objekt*
  - identifizierbare, in sich geschlossene Entitaet
  - bietet einen oder mehrere Dienste nach aussen an
- ein *Dienst* wird durch eine Anfrage (Request) ausgeloest
  - eine Anfrage besteht aus: Operation, Zielobjekt, Parametern und optional einem Kontext
- Objekte werden im Ergebnis von Operationen generiert oder geloescht

# Objektmodell der OMG II

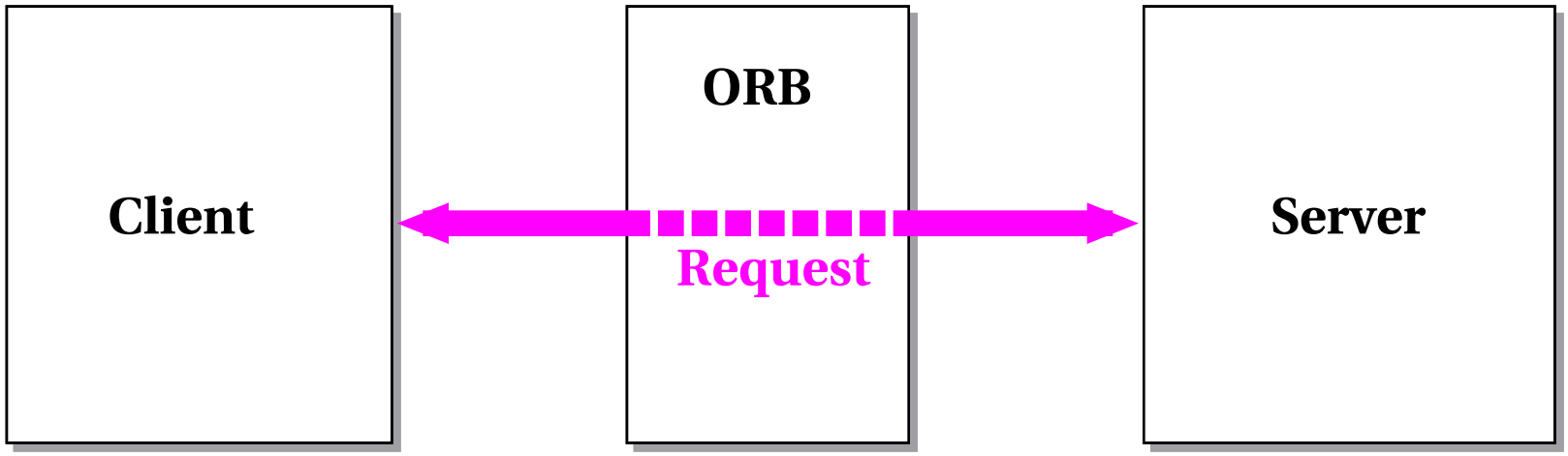
- *Objekttyp*
  - besteht aus der Menge aller Referenzen auf Objekte
- *Schnittstellen (Interfaces)*
  - Beschreibung von Operationen, die ein Klient auf ein Objekt anwenden kann
  - principal interface: transitiver Abschluss (Interface-Ableitungs graph)
  - Interface Spezifikation und Definition sind vollstaendig entkoppelt (Unterschied zum formalen O.-Modell)
- *Attribute*
  - logisch aequivalent: Zugriffsfunktionen

# Objektmodell der OMG III

- Ausfuehrungsmodell
  - Methode
    - Code, der einen Dienst implementiert
    - hat ein Format- Attribut (Menge abstrakter Maschinen)
    - Aktivierung: Ausfuehren der Methode
  - persistente Objekte werden bei einem M.-Aufruf aktiviert
- Objektkonstruktion
  - Mechanismen und Infrastruktur zur Erzeugung und Speicherung von Objekten und deren Zustaenden
  - Implementation: Methodencode und Infrastruktur

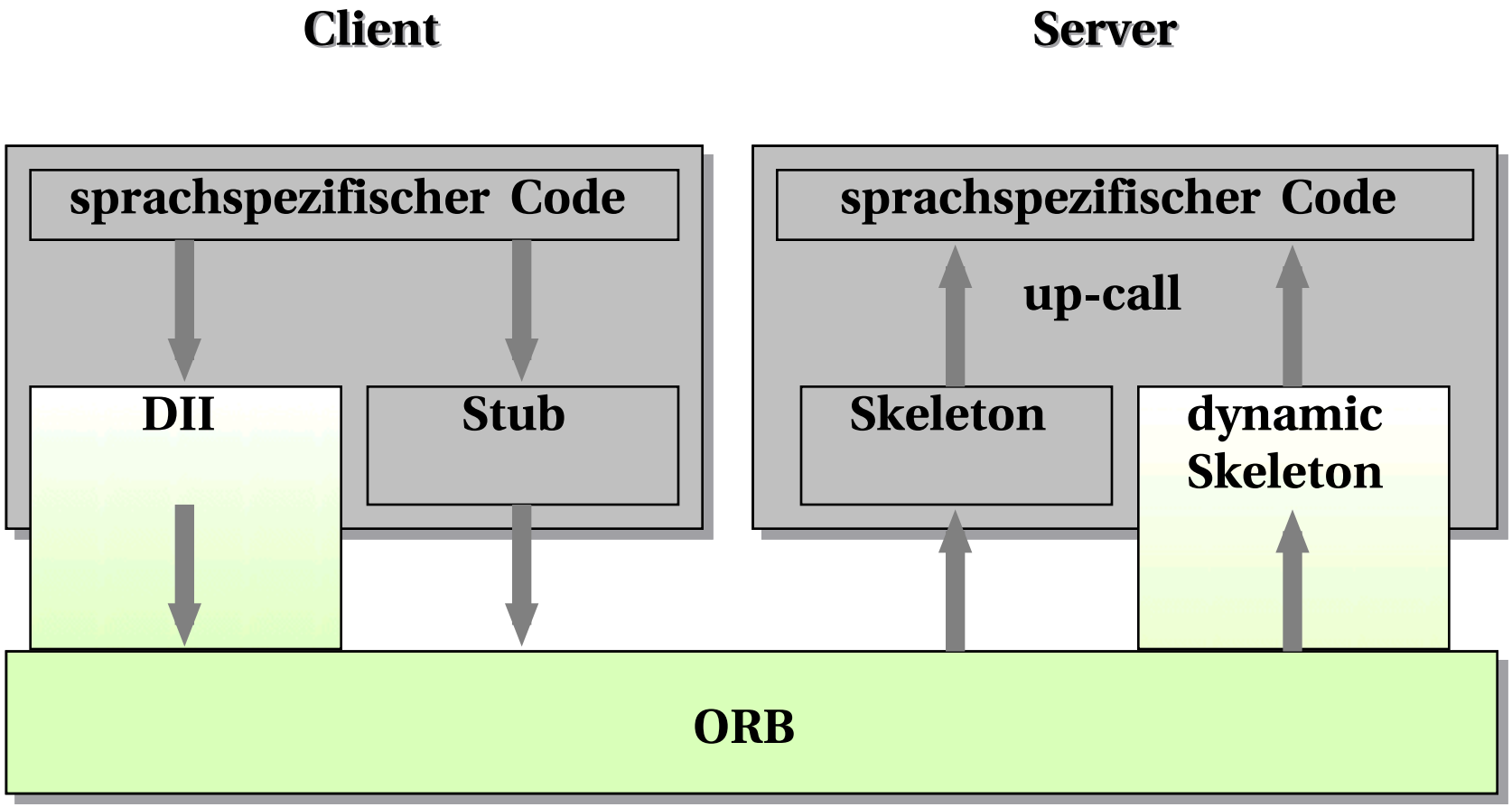
# CORBA I

- CORBA - Common Object Request Broker Architecture
- Object Request Broker




synchroner Methodenaufruf

# CORBA II


Client
Server
sprachspezifischer Code
sprachspezifischer Code
up-call
DII
Stub
Skeleton
dynamic Skeleton
ORB

# CORBA III

```
class MyClass {
  public: virtual void M1()=0;
};
```

```
class MyClass_Stub :virtual public MyClass {
  public:
    MyClass_Stub() { // ORB Konstruktionsanforderung ... }
    virtual void M1() { // ORB Methoden-/Funktionsrufe }
};

class MyClass_Skel :virtual public MyClass {
  public:
    void Dispatcher (char *methodname) {
      // wird vom ORB gerufen, ruft Implementation
      if (!strcmp(methodname,'M1')) M1();
    }
};
```

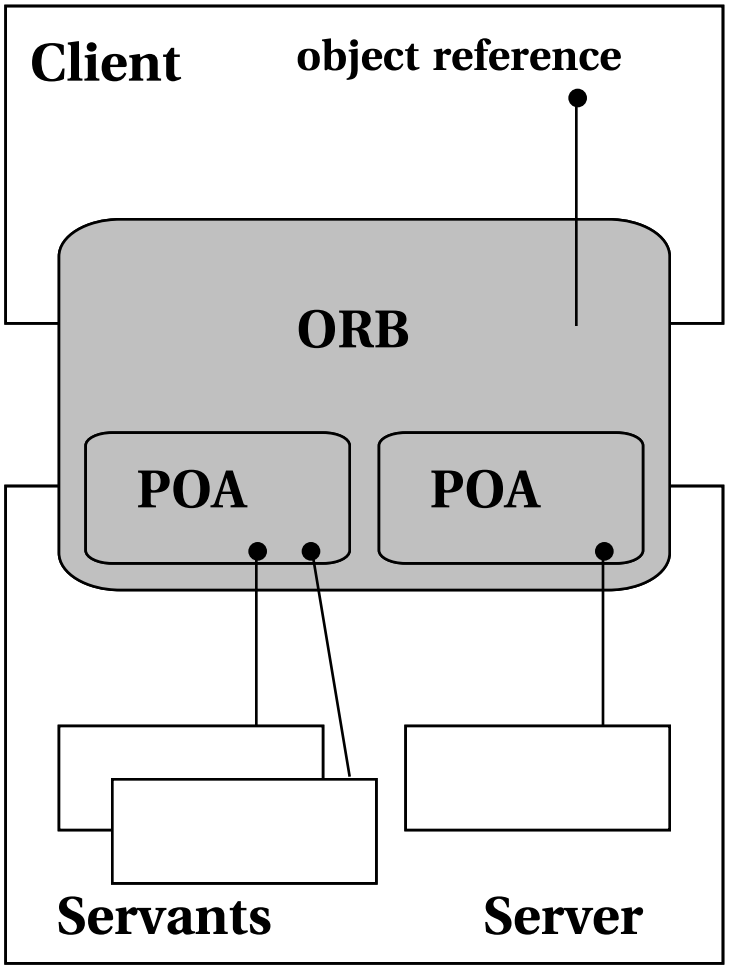
```
class MyClass_Impl :virtual public MyClass_Skel {
  public:
    void M1() { // Implementation ... }
};
```

# CORBA IV

- CORBA definiert eine Menge von Diensten und Schnittstellen
  - Operationen auf Objektreferenzen, Policies, Domain Management Operationen, Thread Management
- *Dynamic Invocation Interface (DII)*
  - CORBA Dienst, Basis aller Methodenaufrufe
- *Dynamic Skeleton Interface (DSI)*
  - ermoeglicht expliziten Zugriff auf O.-Eigenschaften
- *Interface Repository*
  - persistente Ablage von Interfacedefinitionen
  - Zugriff auf Objektdefinitionen (Aktivierung)
  - implementiert ein 'Runtime Type Information' System

# CORBA V

- Portable Object Adaptor
  - 'Erweiterung' des Basic Object Adaptor
  - Basis fuer portable Objektimplementationen
  - unterstuetzt Persistenz von Objekten
  - transparente Objektaktivierung
  - 'multiple Schnittstellen'
  - Unterstuetzung transienter Objekte
  - Policies
  - Aggregation



# CORBA VI

- Interoperabilitaet umfasst
  - ORB-ORB Kommunikation
    - Basis: General und Internet InterOperability Protocol (GIOP/IIOP), DCE-CIOP
    - zwei Konzepte: in-line bridging, request-level bridging
  - ORB-COM/DCOM
    - Datentypanpassung
    - CORBA/DCOM gateways
  - ORB-JavaRMI
    - via IIOP (von SunSoft freigegebenes Protokoll)

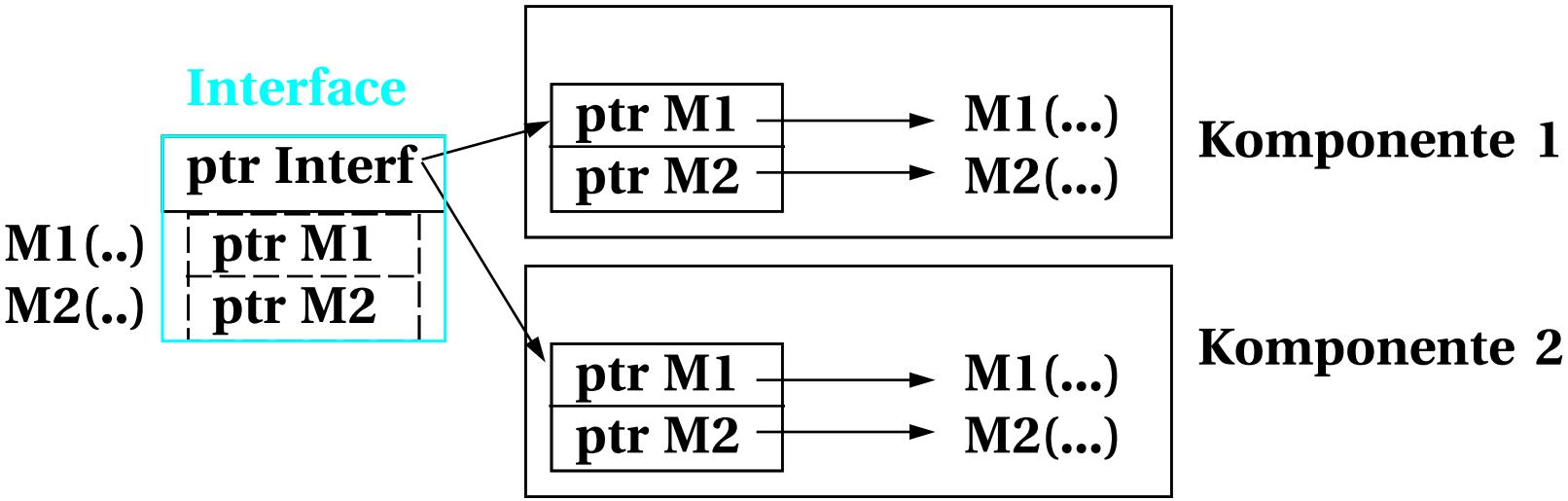
- CORBA-Dienste: 15 Basisdienste; *Naming*, Externalization, *Persistance*, *Events*, *Life Cycle*, Transactions, Properties, Query, Concurrency, Relationship, Collections, Time, Security, Licensing, *Trader*

# MS COM I

- Software Architektur
  - definiert binaeren Standard fuer Komponenten Interoperabilitaet
  - unabhaengig von Programmiersprache
  - multi Plattform (MS Windows, WindowsNT, MacOS, UNIX)
  - dynamisches Laden und Entladen von Komponenten
  - a priori nicht fuer verteilte Anwendungen
  - die Implementation von COM ist nicht offengelegt
- Grundidee
  - vollstaendige Entkopplung von Implementationen und Schnittstelle einer Komponente

# MS COM II

- Schnittstelle und Implementation sind entkoppelt, wenn aus der 'binaeren' Gleichheit zweier Schnittstellen nicht notwendig die Identitaet der Komponenten folgen muss
- COM verwendet Tabellen mit Funktionszeigern
- die Signatur entspricht der Reihenfolge der gespeicherten Funktionszeiger



# MS COM III

- Abbildung auf OO-Sprache
  - Funktionszeigertabelle ist genau die 'virtual method table' (VMT)
  - jede abstrakte Basisklasse mit ausschliesslich 'pure virtual functions' und ohne Attribute kann ein Interface sein
  - die Implementation erfolgt, in abgeleiteten Klassen
- Laufzeit-Polymorphie: trivial
- Erweiterbarkeit im Sinne des objektorientierten Programmierens
  - Hinzufuegen von Funktionszeigern zur Tabelle, z.B. durch Vererbung/Erweiterung

# MS COM IV

- weitere Aspekte
  - a) erweiterte/multiple Schnittstellen zu einem Objekt
    - Problem: in C++ ist eine cast Operation noetig, um korrekte VMT Repraesentationen zu erhalten
  - b) Ressourcen Verwaltung; garbage-collection
- COM Loesung
  - a) Implementation der cast Operation als Funktion bzw. Methode (QueryInterface)
  - b) Zaehlen ausgegebener Referenzen auf die Funktionszeigertabelle

# COM Interfaces I

- I. sind keine Klassen und koennen nicht instanziert werden
  - Die Instanzierung von Objekten (COM Komponenten) erfolgt grundsaetzlich ueber API Aufrufe an die COM Bib.
- I. sind keine COM Komponenten
- Der Zugriff auf Objekte erfolgt ausschliesslich ueber Referenzen auf Schnittstellen (*interface pointer*).
- Eine COM Komponente kann multiple Schnittstellen implementieren.
- Jedes Interface ist global eineutig gekennzeichnet (GUID).
- I. koennen nicht geaendert werden (nach der Freigabe).
- I. koennen nur einfach vererbt werden.

# COM Interfaces II

- **Basisinterface: IUnknown**
- **alle I. muessen von IUnknown abgeleitet sein**

- **QueryInterface**
  - **konvertiert (falls moeglich) den uebergebenen *interface pointer* (ppv) in einen Zeiger auf das angeforderte Interface (riid)**

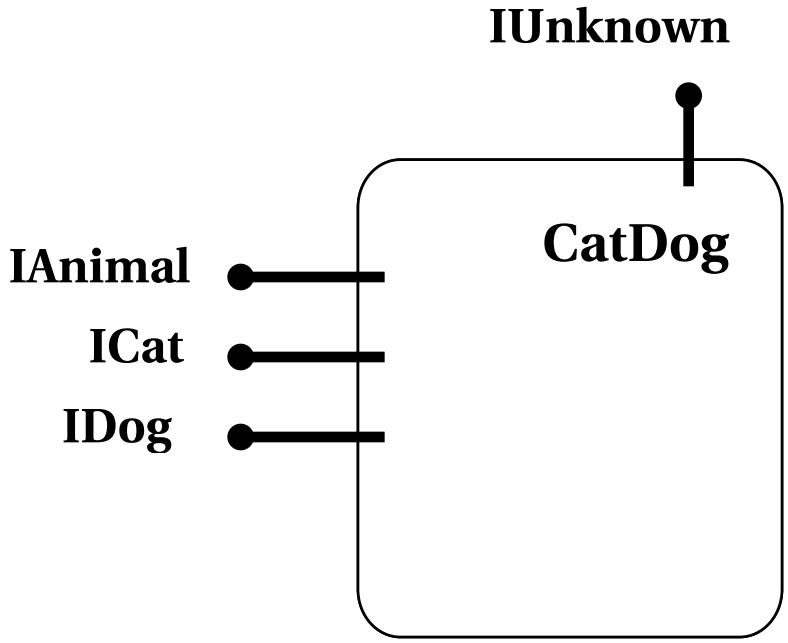
```
[ local, object,
 uuid(00000000-0000-0000-C000- 000000000046) ]
interface IUnknown {
 HRESULT QueryInterface([in] REFIID riid,
 [out] void **ppv);
 ULONG AddRef(void);
 ULONG Release(void);
}
```

# COM Interfaces III

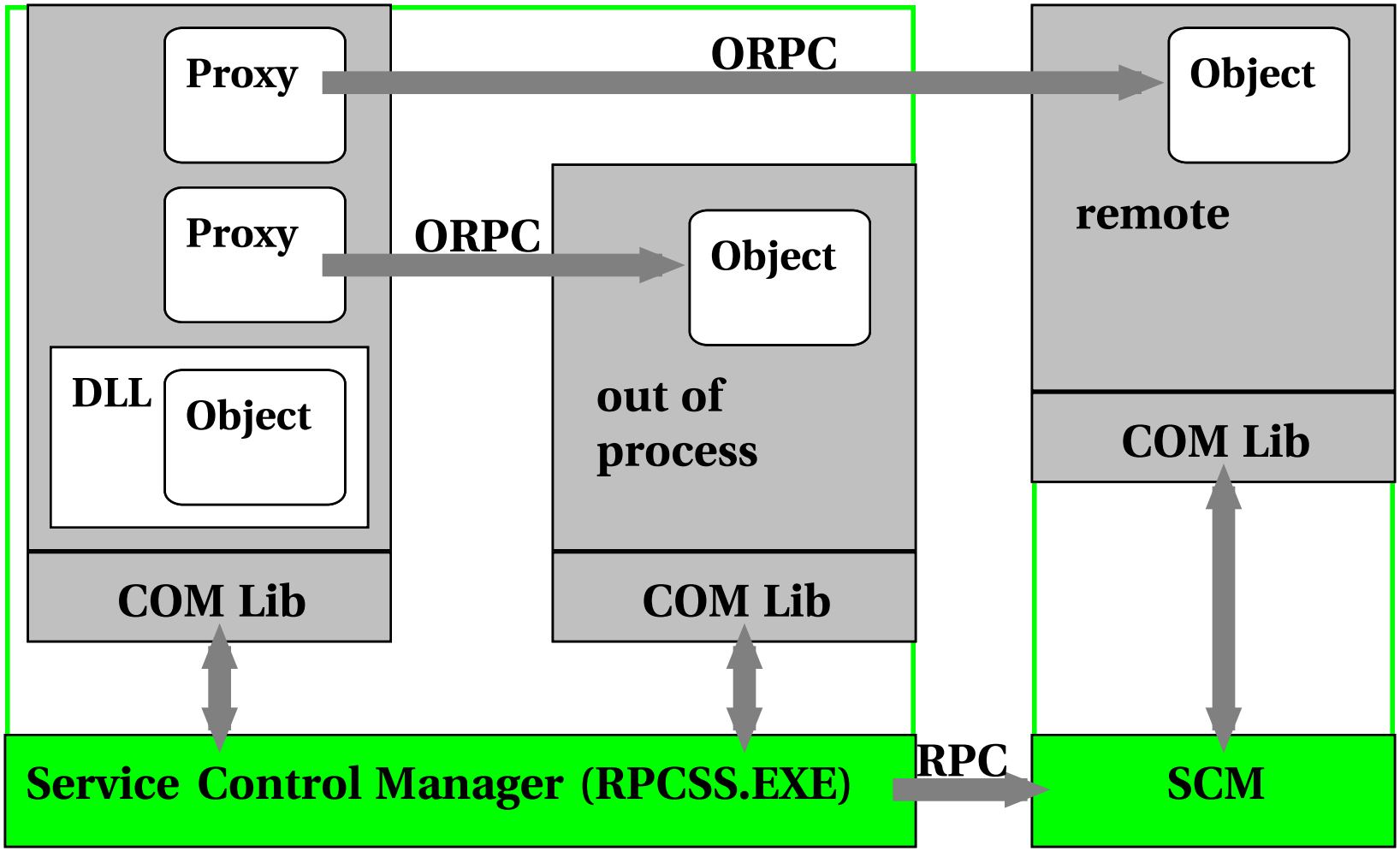
- **Beispiel:**

```
interface       IAnimal:IUnknown        {
  HRESULT   Eat(void);
}
interface       ICat:IAnimal        {
  HRESULT   IgnoreMaster(void);
}
interface       IDog:IAnimal        {
  HRESULT   Bark(void);
}

class     CatDOG
  :public       ICat,     public      IDog    {
  //     ...
};
```

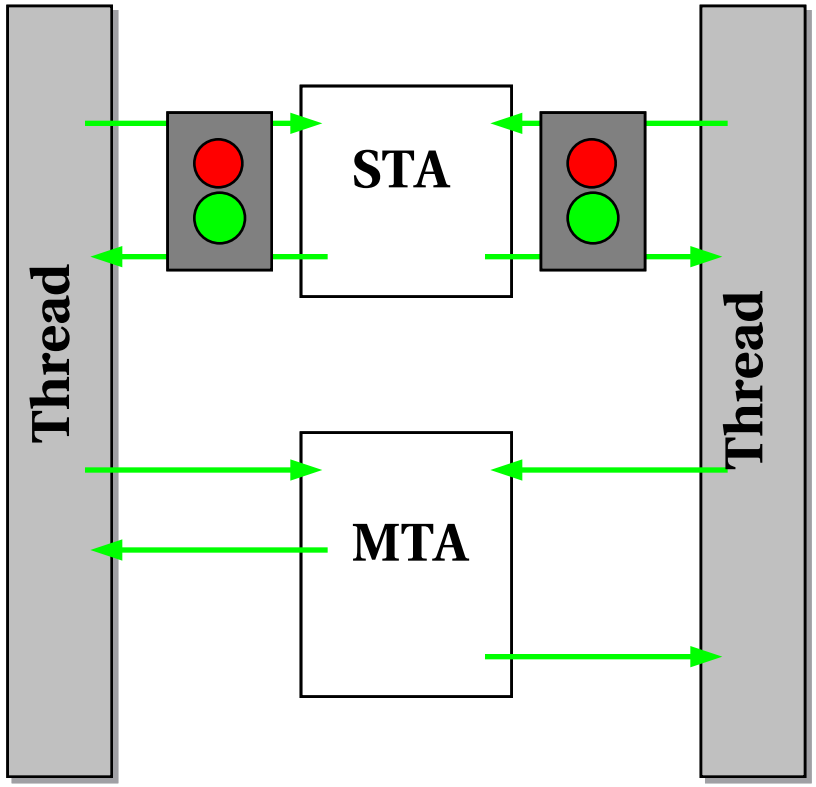


# COM Aktivierungsmodell

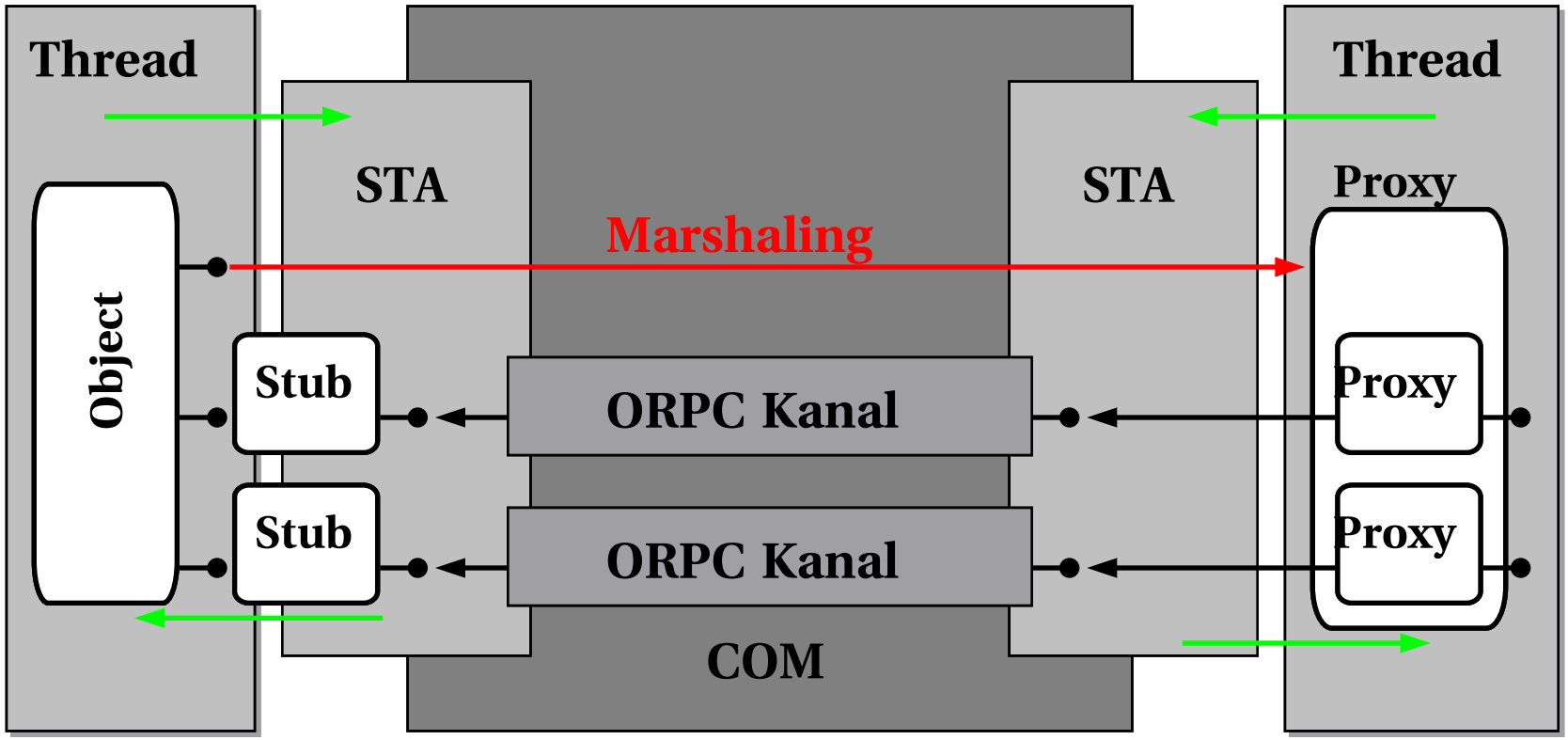


# Distributed COM I

- Apartements
  - -logische Gruppe von Objekten
  - -jedes Objekt gehoert zu genau einem Apartement
  - -mehre O. koennen zum selben Apartement gehoeren
    - -STA; single threaded ap.
    - -MTA; multi threaded ap.
    - -RTA; rental threaded ap.



# Distributed COM II



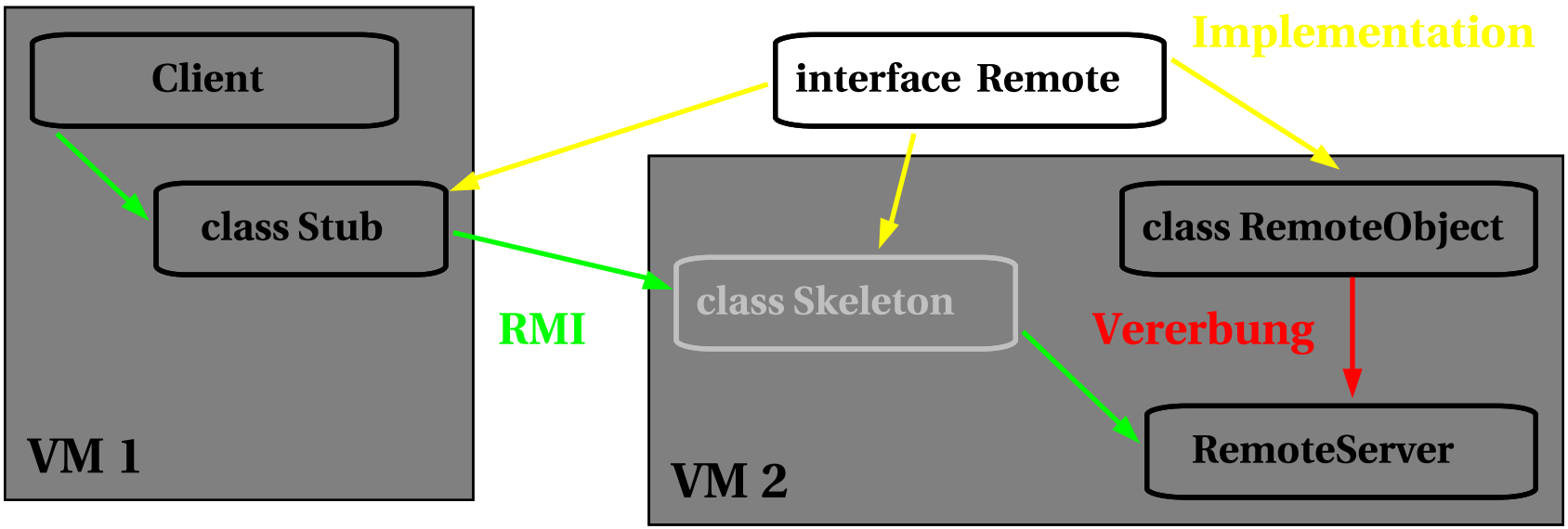
# JavaRMI I

- rein Java-basiertes Schema zur Realisierung verteilter Objekte

- Objektmodell:
  - *Server* generiert Objekte
  - *Klient* nutzt die Objekte des Servers
  - *remote-Objekt* ist ein Objekt, dessen Methoden von einer anderen virtuellen Maschine aus aufgerufen werden koennen
  - remote-Objekte werden durch ein oder mehrere *remote-Schnittstellen* beschrieben
  - *remote method invocation* - Aufruf einer Methode einer remote-Schnittstelle auf einem remote-Objekt

# JavaRMI II

- Implementation



# JavaRMI III

- RMI ab Version 1.1.x
- ab 1.2.x sind keine Skeletons mehr noetig (Stubs und Skeletons werden mit dem 'rmic' Compiler generiert)
- RMI ist RPC basiert und arbeitet wie folgt
  - Verbindungsaufbau zur Ziel- VM
  - synchroner Methodenaufruf
    - Marshaling von Aufruf und Parametern
    - Warten auf Ergebnis
    - Unmarshaling
- JavaRMI enthaelt bereits Klassen, die
  - *dynamisches Laden, Serialization Service, remote Exceptions, Naming-Service* (Registry-Interface), *remote Object Aktivierung* und *remote garbage collection* implementieren

# JavaRMI und CORBA

- bis (exklusive) 1.2.x beherrscht JavaRMI IIOP
- ab Version 1.2.x geben remote Objekte das Ergebnis direkt an den Klienten
- ab Version 1.1.6: RMI-IIOP
  - erlaubt die Wahl zwischen JRMP bzw. IIOP
  - implementiert die Umsetzung call-by-value / call-by-reference
  - Anpassung des Naming-Service

# Vergleich I

- aus 'Business'-Sicht:

| | CORBA | COM/DCOM | JavaRMI |
|---|---|---|---|
| Objektmodell | + | + | + |
| Standard | OMG Spezifikation + | Microsoft Impl. | Sun Impl. |
| Sprachunabh. Srachen | + C++, Smalltalk, Ada95, Java | + C, C++, (VB, Java, Ada) | - Java |
| Plattform | heterogen | NT, UNIX, Mac | unabhaengig |
| Fokus | Enterprise | Desktop | WWW |
| Verfuegbarkeit | verschiedene seit 1992 | Microsoft seit 1996 | Sun seit V 1.1.x |
| Dienste | + | - | - |

# Vergleich II

- aus OOP-Sicht (Systemniveau):

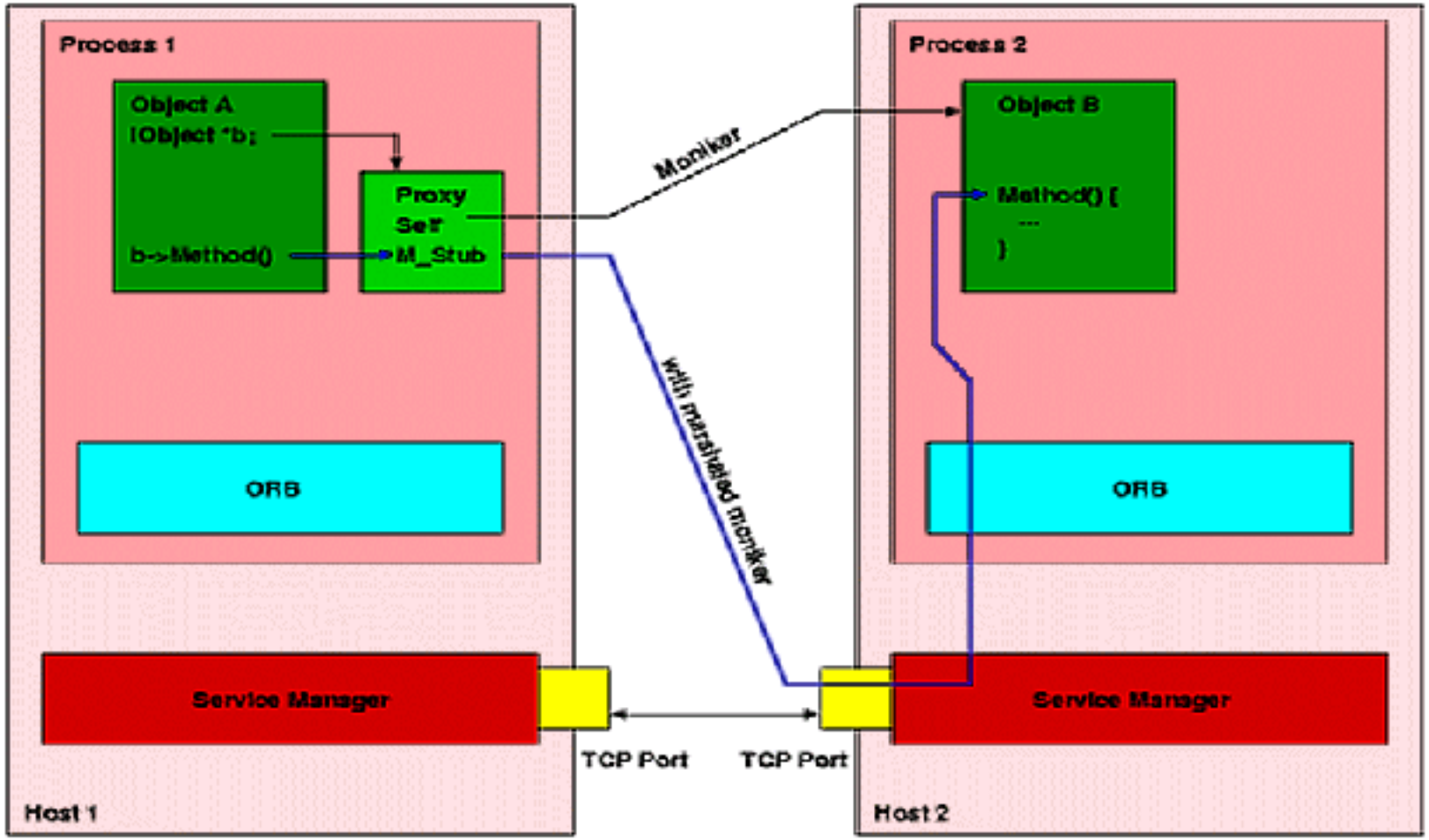
| | CORBA | COM/DCOM | JavaRMI |
|---|---|---|---|
| Kapselung | + | + | + |
| Vererbung | nur Interface | nur Interface | + |
| mehrfach | (+) | (-) | + (Interface) |
| M. Ersetzung | - | - | + |
| M. Verfeinerung | - | - | + |
| M. Delegation | + (Aggreg.) | + (Aggreg.) | + |
| is-a (Klassenhier.) | + (nur Interf.) | + (nur Interf.) | + |
| Polymorphie | + | + | + |
| spaete Bindung | DII/DSI | QueryInterface | + |

# Diskussion I

- Objektmodell
  - Interface = Klassensignatur
  - 'von Signatur abgeleitete Klasse' = Implementation
  - Instanz einer Klasse = 'das' Objekt
  - Referenz = verallgemeinerter Zeiger (Kontext + Referenz)
- Aufgaben eines ORB:
  - Instanzierung von Klassen
  - dynamischen Laden/Entladen von Code (Vererbungshierarchie muss von der Klasse bereitgestellt werden)
  - Kommunikation
- Eigenschaften:
  - Implementationsvererbung *und* Aggregation moeglich
  - Zustandsmodellierung moeglich
  - Ortstransparenz, Migration moeglich

# Diskussion II

- **Objekt-Schema/Ausfuehrungsmodell:**



# Diskussion III

- Instanzierung (in-process):
  - Anforderung einer neuen Instanz einer Klasse vom ORB
  - ORB erbittet Laden des Klassencodes vom Implementation-Repository
    - Impl.-Repository erfragt Elternklassen von zu instanzierender Klasse und laedt deren Implementation
  - ORB instanziert Klasse und bildet 'verallgemeinerte Referenz auf Objekt'
  - Klient erhaelt diese verallg. Referenz
- Instanzierung (out-of process):
  - ORB leitet die Anfrage an *einen* Service Manager weiter
  - Service-Manager startet neuen ORB- Process
    - ORB erhaelt eine in-process Instanzierungsforderung und gibt deren Ergebnis an dem SM zurueck
  - SM sendet diese an den rufenden ORB

# Diskussion IV

- zwingend erforderlich: Strategie zur Verteilung von Instanzen
  - Modellierung des Zustandes
    - Zustandsgraph, gebildet aus den Objekten als Knoten und ihren Relationen
      - use-a = direkte Kommunikation
      - has-a, is-a = Indirektionen, die auf use-a Relationen fuehren koennen
  - statischer Ansatz: a priori Verteilung bei Instanzierung
  - dynamischer Ansatz: checkpoint-restart
  - generelles Vorgehen
    - 'Anhalten' des Objektsystems (checkpoint)
    - Zustandsmodellierung (Vorhersage?)
    - Abbildung auf Ressourcen-Graph (Scheduling)
    - Umverteilung (Migration)
    - 'Wiederanlaufen' (restart)

# Literatur


- 'A Theory of Objects'; M. Abadi, L. Cardelli; 1996
- 'Eiffel: The Language'; B. Meyer; 1992

- 'CORBA 2.0 - Specification'; OMG; 1996
- 'Instant CORBA'; R. Orfali, D. Harkey, J. Edwars; 1998
- 'Inside CORBA'; T. Mowbray, W. Ruh; 1997
- 'CORBA - Standard, Spez., Entwicklung'; A. Sayegh; 1997

- 'Essential COM'; D. Box; 1998

- 'Java Remote Method Invocation Specification'; Sun Microsystems; 1998