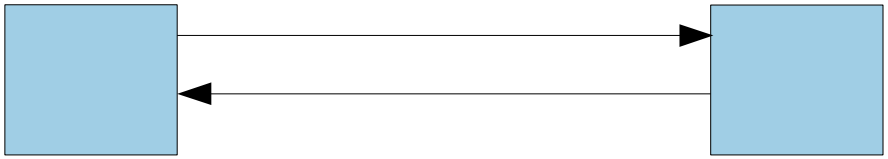
Kommunikation und Migration

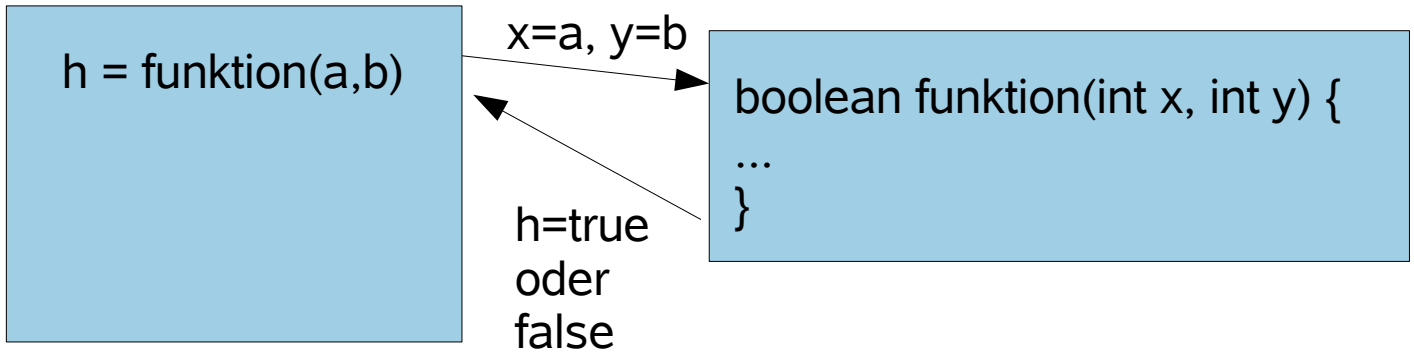
Allgemeine Kommunikationsmuster

- Kommunikation zwischen Komponenten
  - Prozeduraufruf / Methodenaufruf
  - entfernter Prozeduraufruf / entfernter Methodenaufruf
  - gemeinsamer Speicher
  - Nachrichtenaustausch
  - Tuple Space



Prozeduren (lokal)

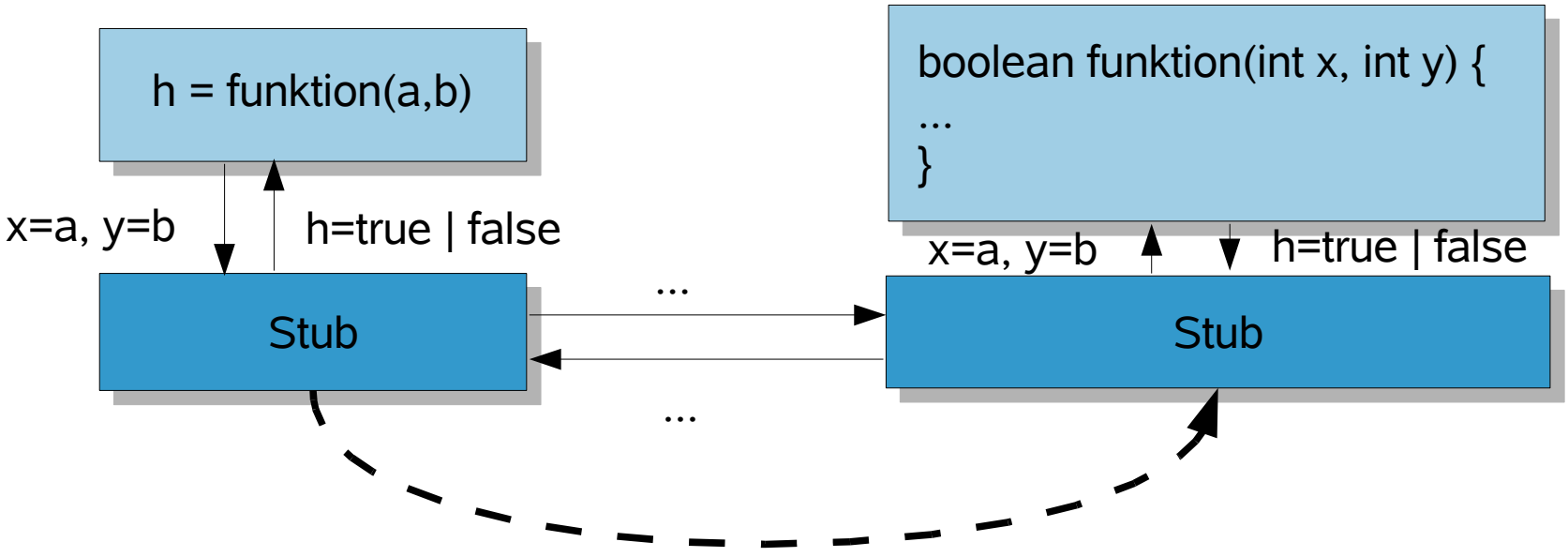
- Informationsaustausch
  - Aufruf einer Prozedur (mit Namen), Übergabe von Parametern
  - Rückgabe eines Wertes



- Bestand der Information
  - dauerhaft beim Aufrufer
  - temporär beim Aufgerufenen (wenn keine Seiteneffekte vorhanden)

Entfernte Prozeduren

- Kenntnis der Lokation des Prozedurservers erforderlich
  - Referenz durch Socketadresse
- Übermittlung der Werte durch „Marshalling“
  - Übersetzung der Werte in ein übertragbares, rückübersetzbares Format
  - Verfahren durch Stubs erleichtert (Verteilungstransparenz)

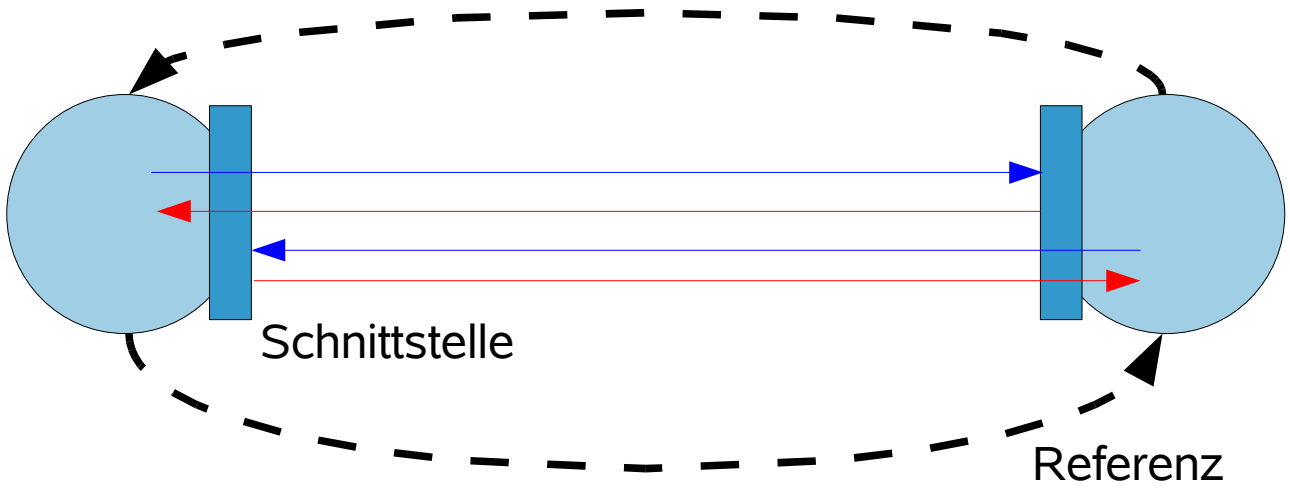


Entfernte Prozeduren

- Bestand der Information
  - dauerhaft beim Aufrufer
  - temporär beim Aufgerufenen
- Ausnahme
  - Über *Context handles* kann Bezug auf früheren Zustand genommen werden

Objekte

- Nachrichtenaustausch durch Methodenaufruf



- Referenz auf Serverobjekt erforderlich
  - Methodenaufruf funktioniert wie Prozeduraufruf

C++:
```
h = objref->methode(a,b);
```

Java:
```
h = objref.methode(a,b);
```

Objekte

- Bestand der Information
  - dauerhaft beim Aufrufer
  - dauerhaft beim Aufgerufenen (Zustandsänderung!)

Methodenaufrufe sind eine mögliche Form der Kommunikation zwischen Objekten.

- Vorteile der Methodenaufrufe
  - Kommunikation auf Programmiersprachenebene (Verifikation durch Compiler)
  - sehr effizient umsetzbar, sehr effizient ausführbar
  - sehr leicht zu erlernen und zu durchschauen

Objekte

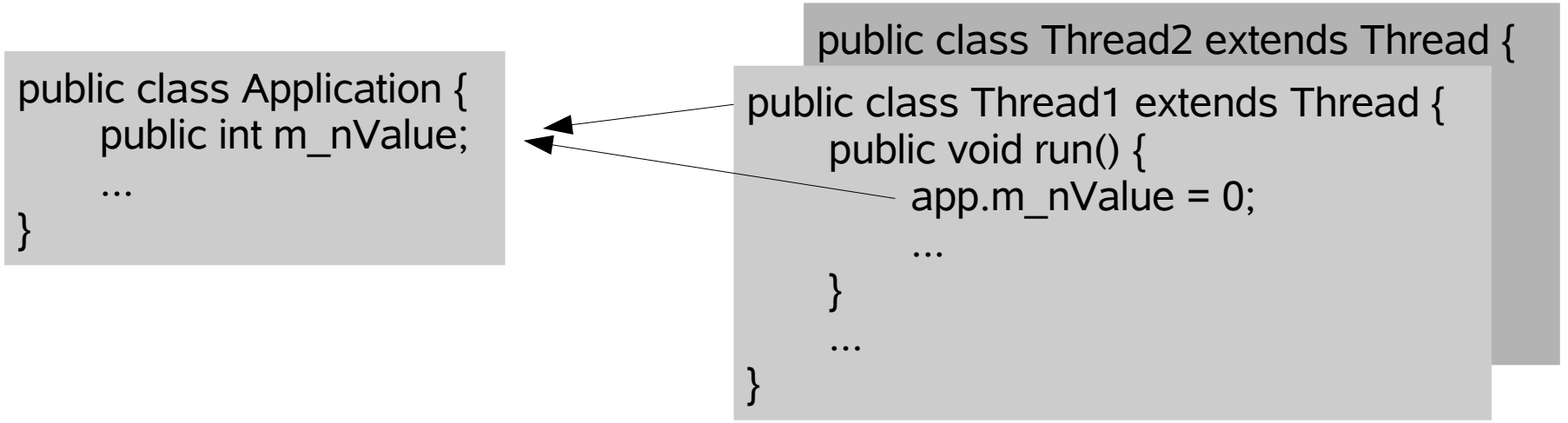
- Nachteile der Methodenaufrufe
  - sehr maschinennah, festgelegte Aufrufsemantik
    - was von der Programmiersprache nicht vorgesehen ist, geht nicht
  - Erfordernis, gültige Referenzen zu verwenden
    - aber im verteilten Falle sind Referenzen nicht immer gültig
  - Direkter Aufruf
    - Schwierig bezüglich Sicherheit (Zugriffskontrolle)
    - Schwierig bezüglich Autonomie (was bedeutet dies für das autonome Objekt?)

Entfernte Objekte

- Ähnliche Situation wie bei entfernten Prozeduren
- CORBA
  - Objektreferenzen
    - zu übersetzen in sprachspezifische Referenzen
    - Stubs / Skeletons
    - IOR (Interoperable OR für querverbundene ORBs)
  - Broker-Funktionalität
    - ORB leitet Anfrage an entsprechendes Objekt weiter
    - Auch Vermittlung möglich (suche Objekt mit bestimmter Schnittstelle)

Gemeinsamer Speicher

- bekannt als „Shared Memory“
- verwendet bei Prozessen, die miteinander kommunizieren sollen
  - in Java: Threads (stets im gleichen Datensegment, da gleicher Prozess)
  - Threads können über globale Felder kommunizieren



Gemeinsamer Speicher

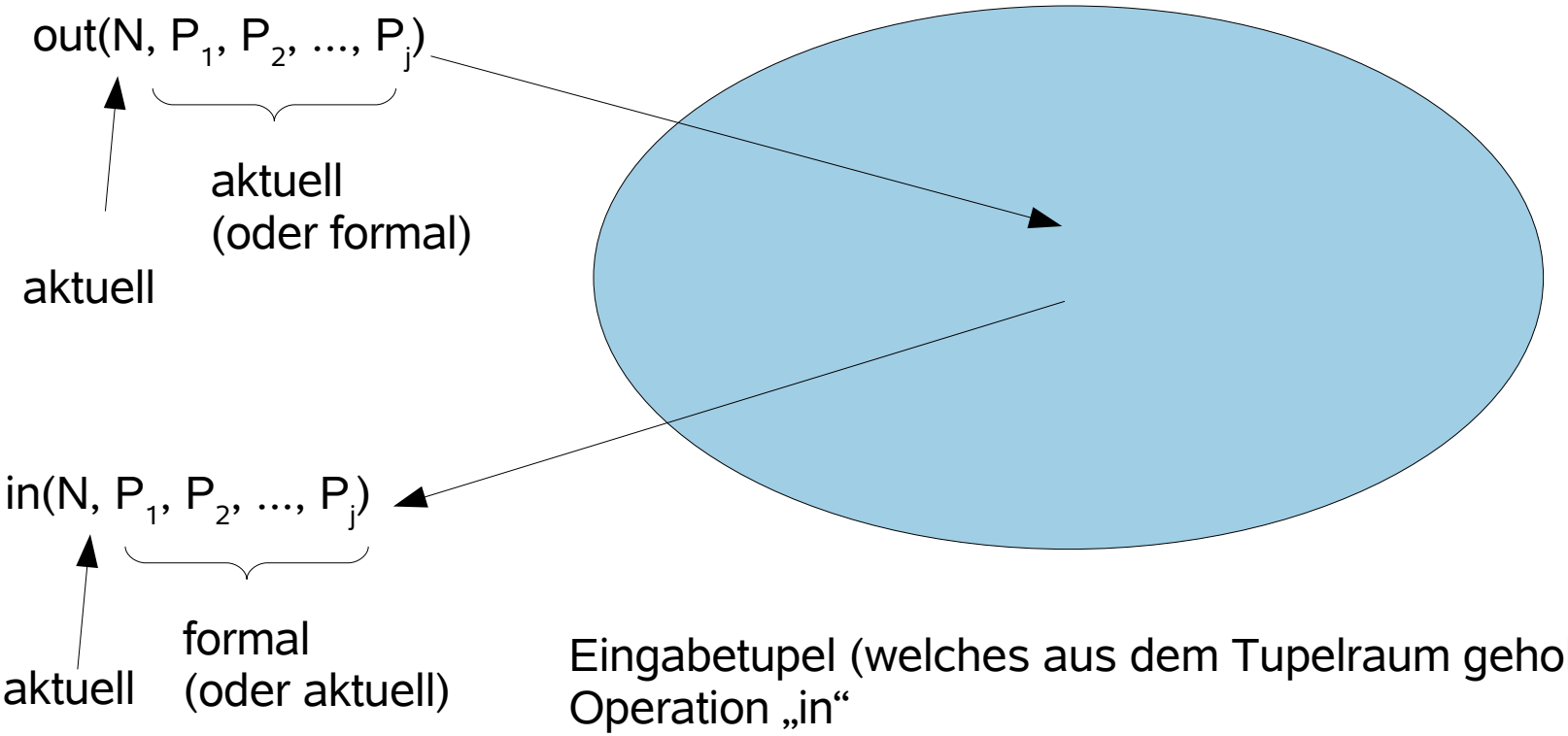
- Vorteil des gemeinsamen Speichers
  - extrem schnell (kein zusätzlicher Aufwand)
  - typsicher (Compiler führt Prüfung durch)
  - leicht zu programmieren
- Nachteil des gemeinsamen Speichers
  - Konkurrenzprobleme (Schutz durch Semaphore erforderlich)
  - nur im gleichen Adressraum (keine entfernte Kommunikation möglich)
    - was wiederum für mobile Agenten spräche (siehe auch „Messengers“)
  - Fehleranfälliger und schlecht zu wartender Code

Tupelraum (Tuple space)

- Koordination paralleler Prozesse ohne direkte Adressierung
- Adressierung anhand Inhalte der abgelegten Tupel
- Idee: Im Raum *zirkulierende Objekte*
  - Auflösung des Klient-Server-Paradigmas
- Tupelraum
  - gemeinsam verwendeter Speicher
  - sogar auf verschiedenen Knoten (Infrastruktur sorgt für Abgleich)
- Tupel bestehen aus aktuellen oder formalen Parametern
  - Vergleich komponentenweise

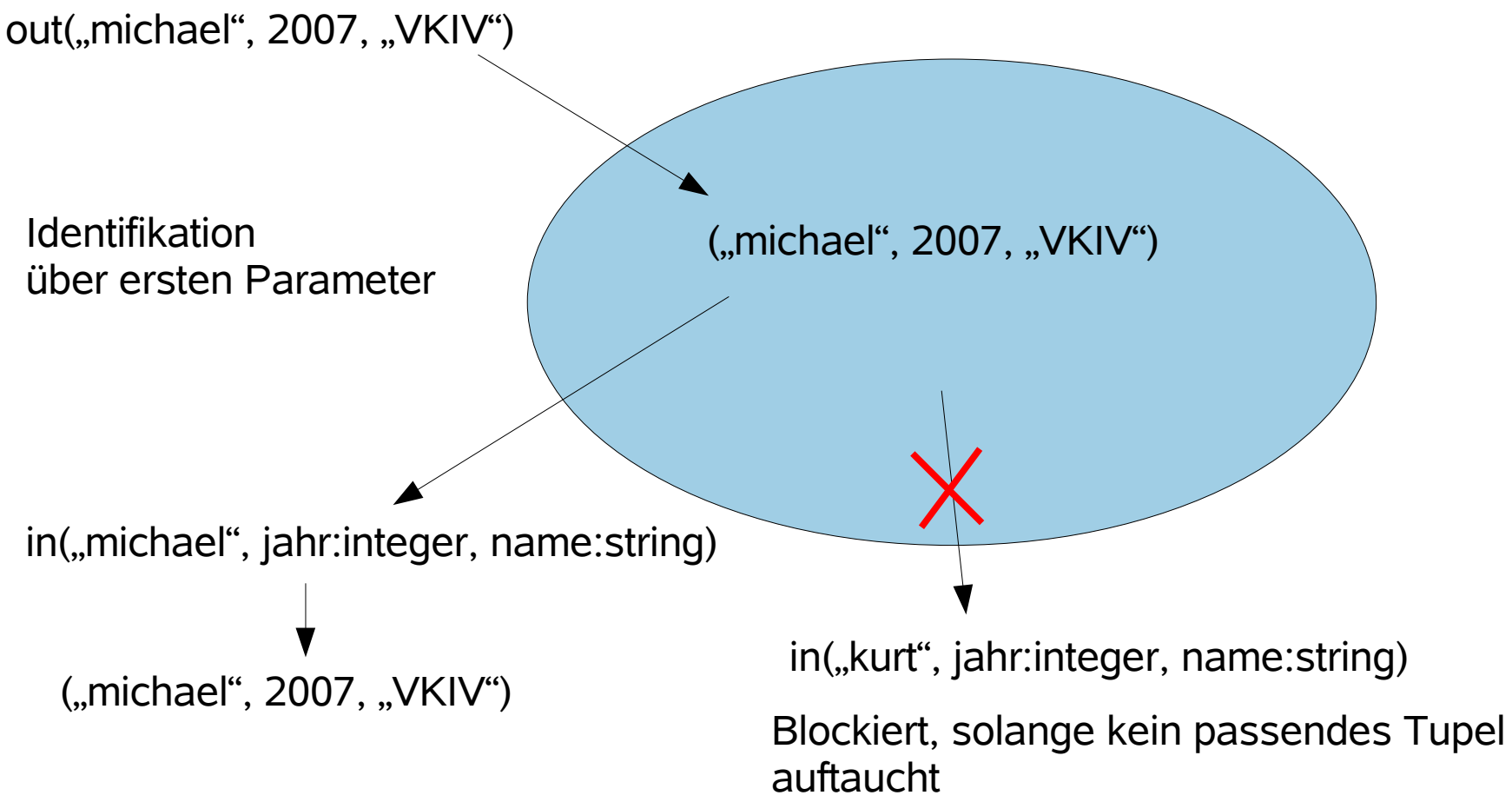
Operationen

Ausgabetupel (welches in den Tupelraum gelegt wird)
Operation „out“



Eingabetupel (welches aus dem Tupelraum geholt wird)
Operation „in“

Vergleich zweier Tupel

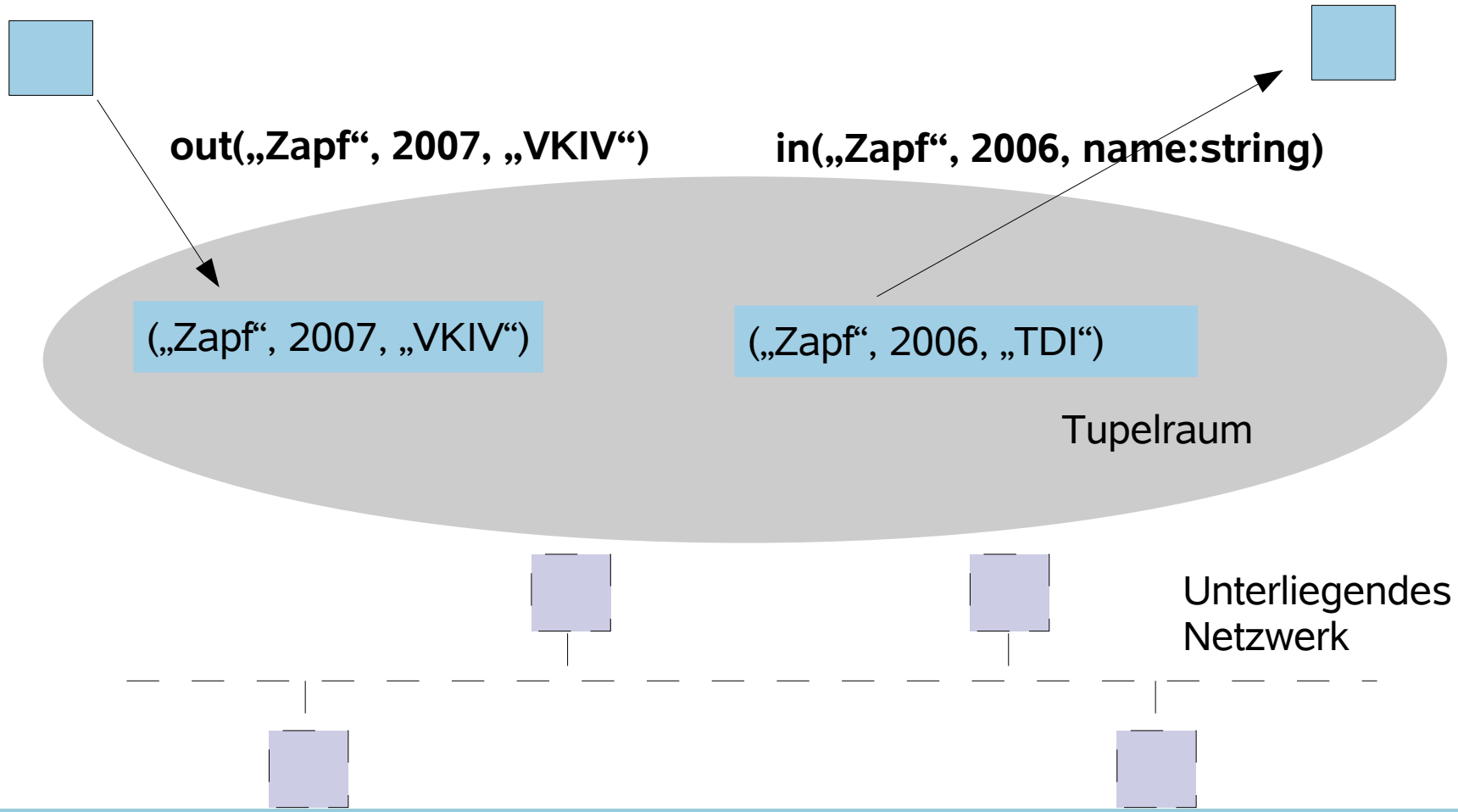


Tupelraum

- Weitere Möglichkeiten
  - In-Parameter aktuell statt formal: Genauere Auswahl
    - z.B. in(„michael“, 2007, name:string): Aktuelle Parameter werden miteinander verglichen
  - Out-Parameter formal statt aktuell
    - out(„michael“, jahr:int, „VKIV“) wird von in(„michael“, 2007, name:string) gefunden
  - Formale Parameter können nicht miteinander verglichen werden
    - „i:integer“ passt nicht auf „j:integer“
- Operation read
  - Wie in, kopiert aber das Tupel (es bleibt im Tupelraum)

Tupelraum

- Abstraktion von Netzwerktopologie

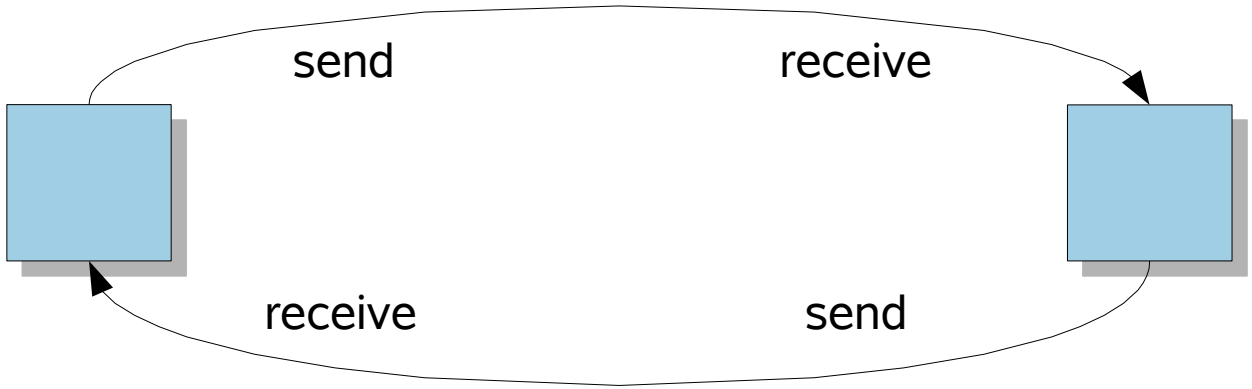


Tupelraum

- Vorteil: Einfach modellierbare, parallele Bearbeitung
- Realisierungen
  - Linda: David Gelernter, Yale
  - JavaSpaces
- Ähnliches Prinzip: JMS
  - Java Message Service
  - Teilnehmer registrieren sich am JMS-Server
  - Teilnehmer veröffentlichen (**publish**) Daten
  - andere können bestimmte Daten anhand von Themen (**topics**) abonnieren (**subscribe**) und entgegennehmen (**consume**)

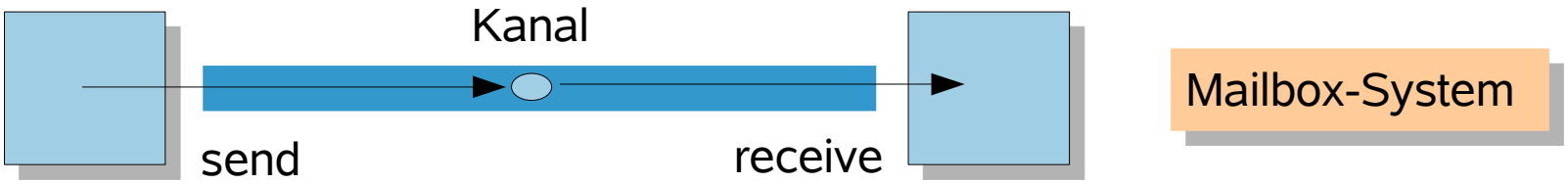
Nachrichten

- „Message passing“
- Verwendung in Interprozesskommunikation
  - kann auch im verteilten Falle verwendet werden
- Spezielle Befehle („Primitive“) in Programmiersprache und in Betriebssystemkern (Systemaufrufe)



Nachrichten

- Kanalkapazität



  - Kanal fasst keine Nachricht
    - **send** blockiert so lange, bis **receive** Nachricht übernimmt
  - Kanal fasst n Nachrichten
    - n * **send** möglich, bevor **send** blockiert
  - Kanal fasst beliebig viele Nachrichten
    - beliebig viele **send**-Vorgänge (**asynchrone Kommunikation**)
    - kommt in der Praxis so nicht vor
    - bei ausreichender Puffergröße praktisch asynchron

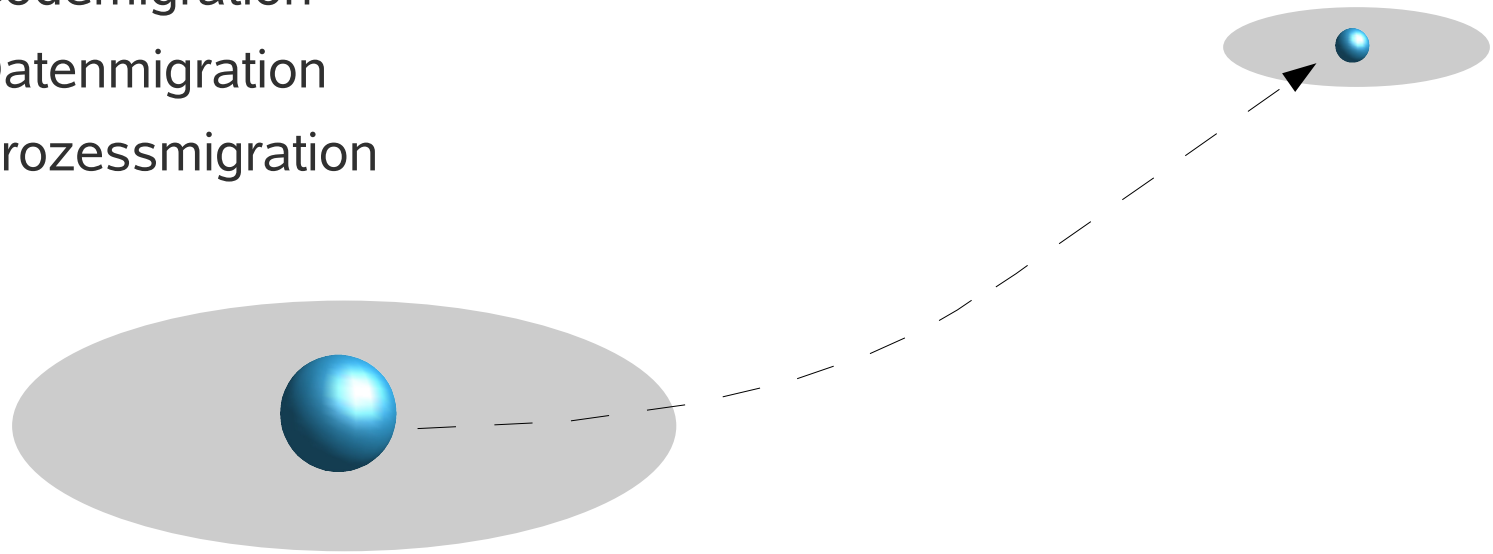
Nachrichten

- Besonderheiten
  - Nachrichten können verloren gehen (ins. bei verteilten Prozessen)
  - Adressierung der Prozesse?
  - Authentifikation?
- Bestätigungsprotokoll zwischen Sender und Empfänger
  - erhöht den Aufwand

Der Methodenaufruf eines Objekts auf einem anderen kann als Nachrichtenaustausch betrachtet werden.
Objekte schicken sich also Nachrichten.

Migration

- Vorgang des Wechsels der Ausführungsumgebung
  - *(im/e)migrare* (lat.) - (ein/aus)wandern
- Verschiedene Formen
  - Codemigration
  - Datenmigration
  - Prozessmigration

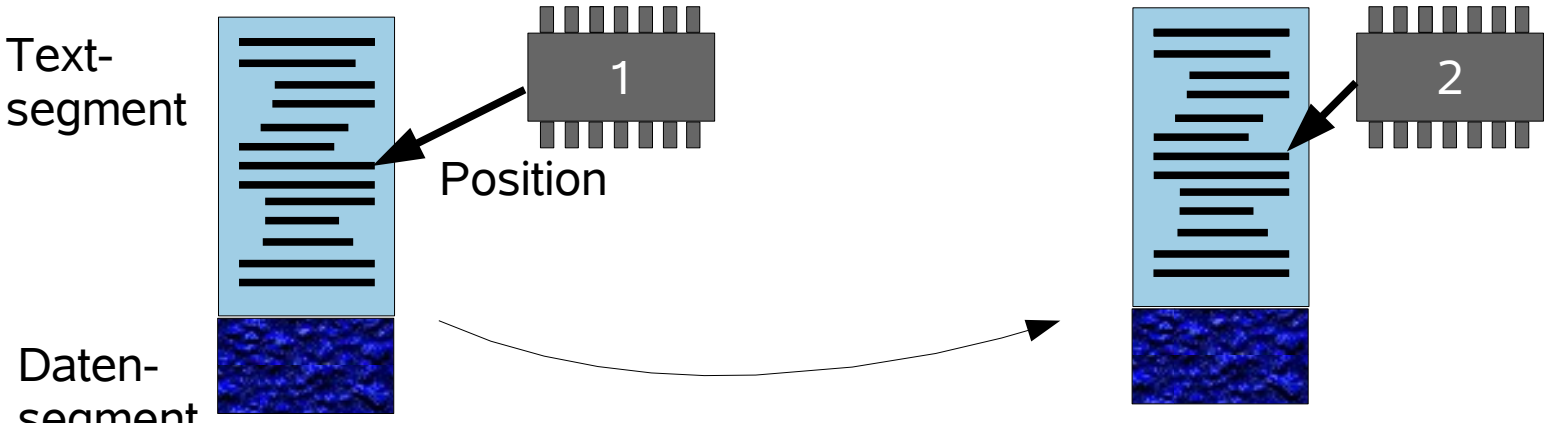


Code- und Datenmigration

- Verlagerung von Code auf einen anderen Netzknoten
  - siehe Code on demand, Remote Evaluation, Mobile Agenten
  - Migrierender Code dient der Funktionsbereicherung der empfangenden Lokation
    - ändert deren Zustand und Verhalten
    - nur bei Agenten wird der Zustand des migrierenden Codes bewahrt
- Datenmigration
  - Verlagern von Daten zu einem anderen Netzknoten
  - Standardvorgang bei verteilten Systemen (Klient-Server usw.)
  - zu beachten: Daten müssen an entfernter Stelle interpretierbar sein
    - ggf. Klassen nachladen (Codemigration bei Bedarf)
  - zu unterscheiden: Felder, lokale Variablen und Ressourcen!

Prozessmigration

- Konzept verteilter Betriebssysteme
  - Verlegen eines Prozesses auf einen anderen Knoten oder Prozessor
  - Komplette Prozessinformationen werden am Ziel wiederhergestellt
- Threadmigration: Wiederherstellung aller Threads
  - inklusive deren Zustände



Einsatz bei Lastverteilung

Passive und aktive Mobilität

- Mobilität: In der Lage sein, Migrationen durchzuführen

Passive Mobilität

Wechsel der Umgebung geschieht nicht „freiwillig“

Betriebssystem verlagert Objekt

Einsatz: Lastverteilung

Aktive Mobilität

Wechsel der Umgebung geschieht „freiwillig“, definiert durch das Verhalten des Objekts

Objekt verlangt nach Verlagerung

Einsatz: Mobile, planvolle Agenten

Passive Mobilität

- Betriebssystem verlagert Objekt / Prozess
- Besonderheiten
  - Definition des Objekts beinhaltet diese Verlagerung nicht
  - Keine Vorkehrungen, um Zustand selbst zu sichern
  - Keine Manifestation innerhalb der Verhaltensbeschreibung
  - ggf. ereignisorientierte Behandlung möglich

Das Betriebssystem muss eine Prozessmigration durchführen
und alle Daten inklusive dem Ausführungszustand wiederherstellen

Passive Mobilität und Agenten

- Für mobile Agenten ist passive Mobilität kaum geeignet
  - Agent muss mit unvorhergesehenen (da nicht in seinem Plan inbegriffenen) Ortswechseln rechnen
  - Kein Verlass, dass die Kommunikationspartner noch erreichbar sind
- Bedingung
  - Agent muss vom Ausführungsort abstrahieren („mir egal, wo ich arbeiten soll“)
  - aber ist das im Sinne von mobilen Agenten?

In diesem Falle ist die Mobilität ein bezüglich des Agenten externes Konzept (d.h. es findet keinen Niederschlag in seiner Programmierung)

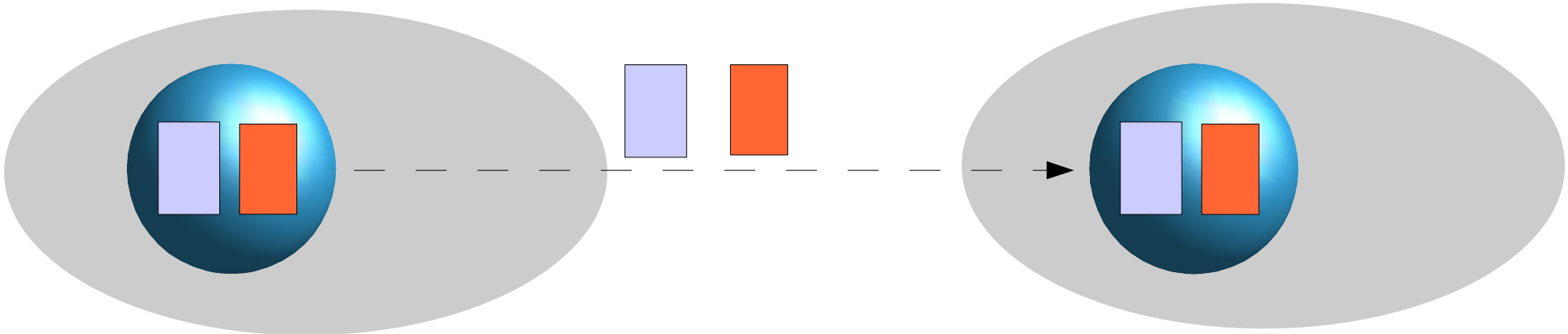
→ Letztlich einfache, autonome Agenten, die an unterschiedlichen Stellen abgearbeitet werden.

Aktive Mobilität

- Objekt verlangt Verlagerung
  - Konsequenzen der Verlagerung sind bekannt und müssen im Code behandelt werden
    - beispielsweise das Abmelden von bisherigen Kommunikationspartnern
    - Sichern des Zustands
- Infrastrukturelle Unterstützung
  - Betriebssystem / Infrastruktur erledigen den Transport
  - ggf. auch Sicherung des Zustands
  - Ausführungszustand retten

Technische Betrachtung der Migration

- Wie läuft die Verlagerung ab?
  - Programmcode muss an die entfernte Position gebracht werden (wenn nicht dort schon vorhanden – dann genügt ein Verweis)
  - Bei mobilen Agenten (MA) muss der jeweilige Zustand ebenfalls an die entfernte Position gebracht werden (sonst REV oder CoD)



Technik der Migration

- Code wird als unveränderlich aufgefasst
  - sonst Teil des Zustands (z.B. modifizierbare Skripte)
- Zustand wird strukturell vom Code vorherbestimmt
  - nicht jedoch vom Inhalt
- Problem
  - Der Zustand existiert in den meisten Programmmiersprachen nicht als eigentliches Objekt
  - lediglich Zugriff auf Komponenten des Zustands (Felder, auch komplexe Objekte)
  - Es gibt daher in der Regel keine expliziten Befehle zur Handhabung des Zustands an sich

Technik der Migration

- Unterstützung der Programmiersprache
  - Konzept des Zustandstransports normalerweise nicht in der Programmiersprache vorhanden

```
connection.sendState(this.getState());
```

wäre wünschenswert

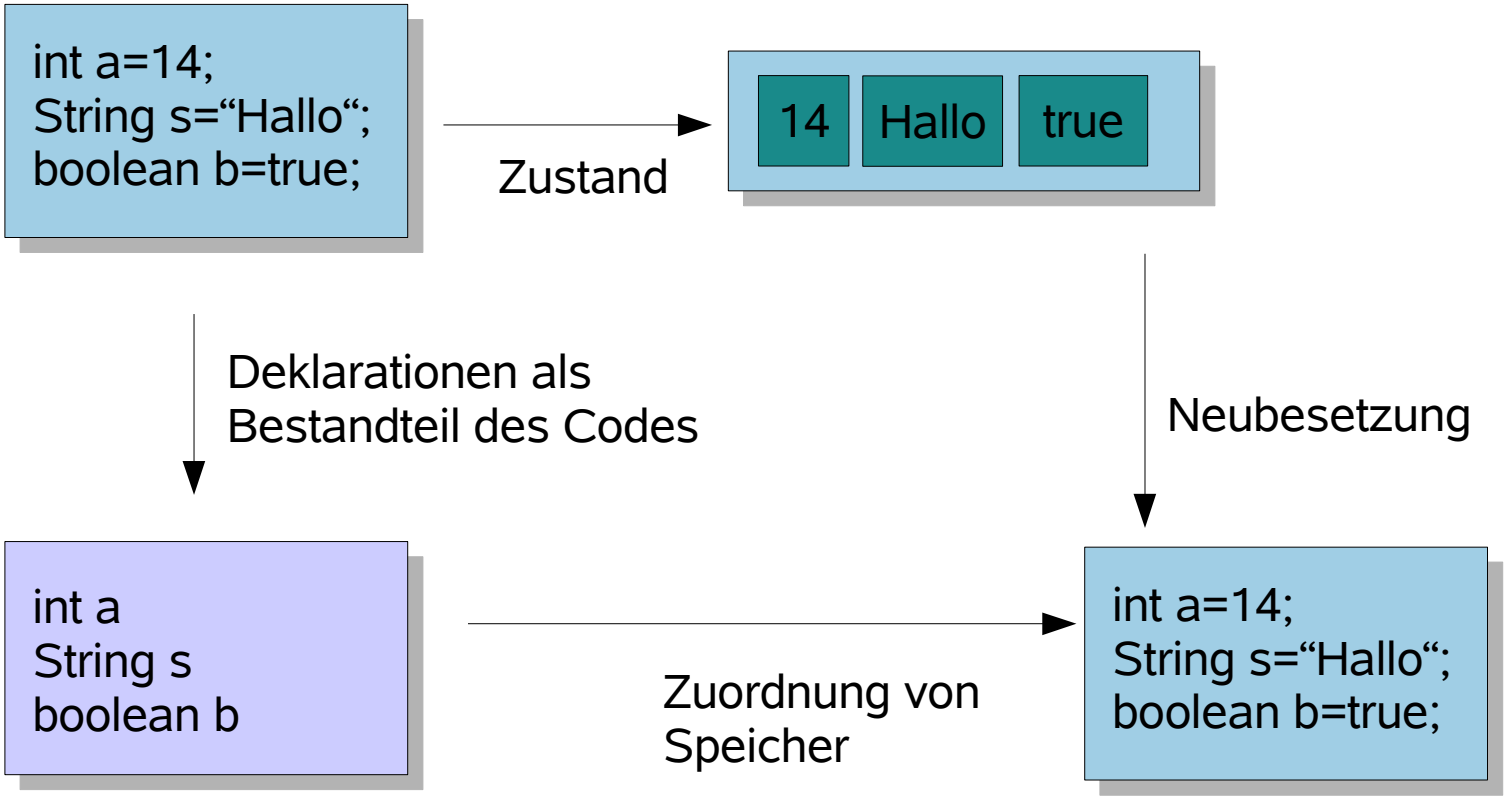
  - d.h. man muss auf Anwendungsebene für die Zustandsbewahrung sorgen und dabei die verfügbaren Transportmechanismen nutzen
- Verfügbare Transportmechanismen basieren meist auf Byte-Transport
  - aber die Handhabung findet auf abstrakter Ebene statt
    - Strings, Zahlen, mehrdimensionale Felder

Migration und Zustand

- Passive Mobilität
  - Zustand wird „extern“ betrachtet – als Datensegment des Prozesses
  - Keine Abstraktion: Datensegment ist ein Adressbereich – also Kette von Bytes
  - Nutzung von Betriebssystemfunktionen (über Systemaufrufe)
- Aktive Mobilität
  - Einfrieren des Zustands problematisch (da interne Migrationsauslösung)
  - Wie wird der Zustand in eine transportable Form gebracht?

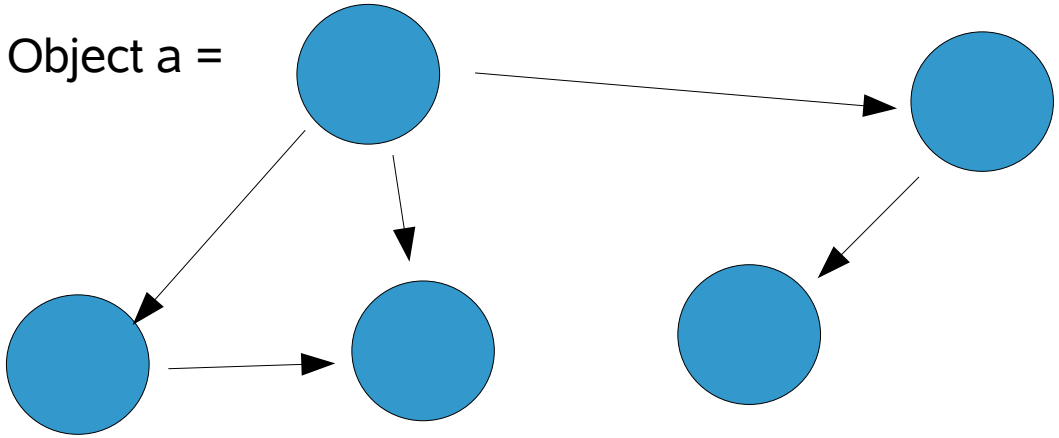
Migration und Zustand

- Deklarationen und Werte



Migration und Zustand

- Die Welt ist komplizierter geworden!



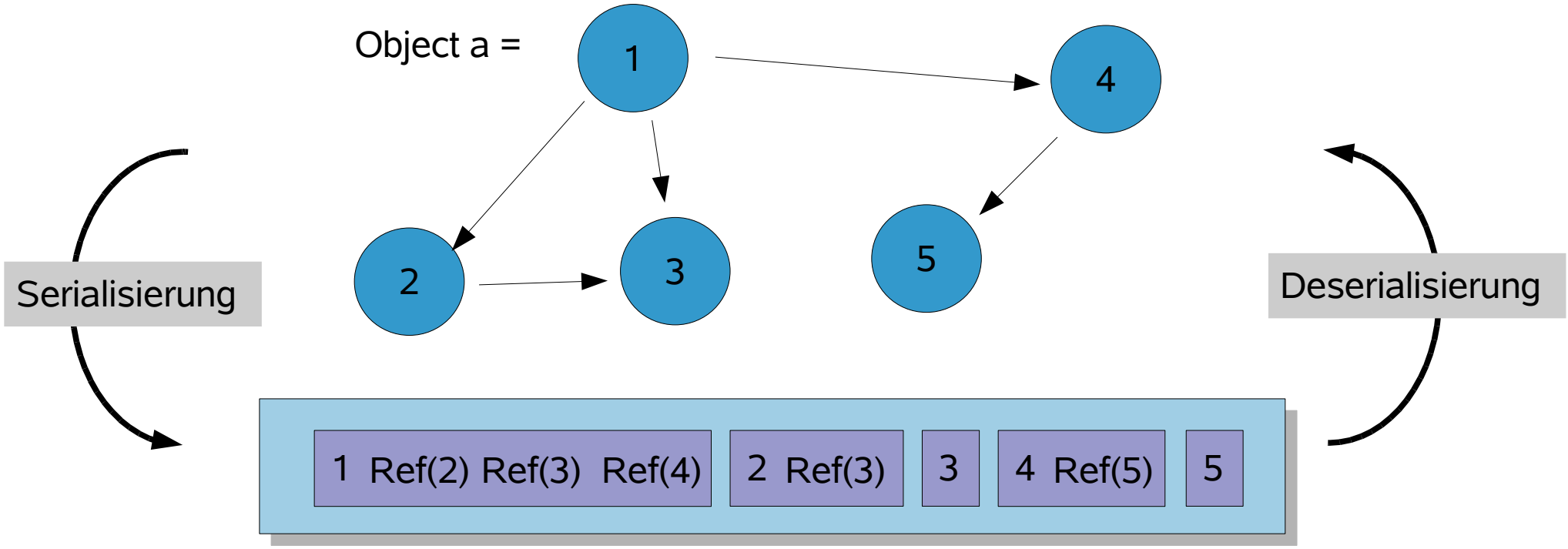
  - Beliebig strukturierter Objektgraph
  - Referenzen zwischen den Objekten
  - Referenz in a-Variable nur auf dem aufspannenden Objekt!

Serialisierung

- in Java-Programmiersprache (und -umgebung)
- Konzept der Serialisierung
  - Zustand kann in einem linearen Format (seriell) gespeichert werden
- Für Applets praktisch unerheblich
  - warum? (siehe CoD, REV, MA!)
- Einführung in JDK 1.1
  - im Kontext der RMI-Einführung (warum dort?)
- Zuvor: Verschiedene Arbeiten zur Bewahrung des Zustands
  - Hohl: Diplomarbeit, Uni Stuttgart
  - daraus entstanden: **Mole**, erstes bedeutendes MAS auf Java-Basis

Serialisierung

- Serielle Form durch Traversieren des Objektgraphs
- Eindeutig rekonstruierbar



Migration unter Java

- Nahe liegender Mechanismus
  - Klassen liegen als Bytecode-Dateien vor
  - Zustand serialisieren
  - Übertragen der Klassen (wie bei Applets)
  - Übertragen des Zustands
  - Laden der Klassen
    - Klassenlader-Konzept (ermöglicht Bereitstellung neuer Klassen zur Laufzeit)
  - Deserialisierung des Zustands
- Aber es fehlt...

Migration unter Java

- ... der Ausführungszustand!

> Die Java-Laufzeitumgebung der Versionen bis 1.6
> ist nicht in der Lage, den Ausführungszustand zu sichern

- Hintergrund
  - Der Ausführungszustand ist eine „innere Angelegenheit“ von Objekten
  - Der Zugriff auf den Ausführungszustand bricht die Abgeschlossenheit der Objekte und deren Interaktion über ihre Schnittstelle

Migration unter Java

- Aber auch handfeste technische Gründe
  - Ausführungszustand hängt von der Zahl aktiver Threads ab
  - Alle Thread-Rahmen müssen gesichert werden (beinhaltet insbesondere die Werte lokaler Variablen)
  - Wiederbeginn der Ausführung kann nicht durch Sprachelemente formuliert werden
    - Wie würde man die Position „in“ einer Methode angeben wollen?
  - Also nur Realisierung als undurchsichtige, spezielle Systemfunktion
  - Lohnende Anwendungsszenarien außer Migration?

Vielleicht eine künftige Erweiterung?

Starke und schwache Migration

- Starke Migration

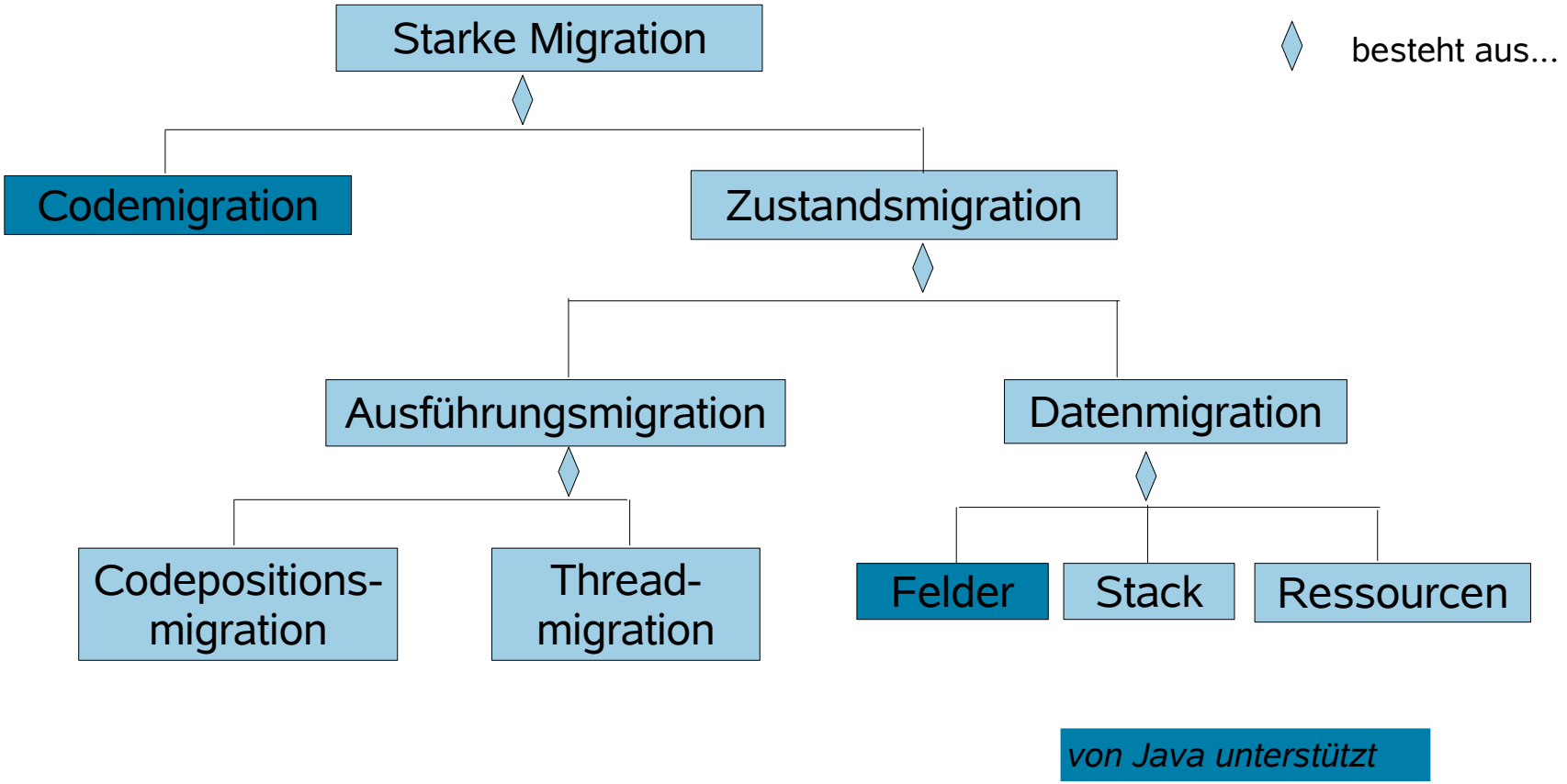
Unter einer **starken Migration** versteht man eine Migration eines Agenten, die den Agenten an der Zielposition in Bezug auf seinen Zustand und seine Beschreibung als präzise Kopie des Ursprungsagenten erzeugt und es ihm ermöglicht, seine Ausführung exakt an jener Stelle fortzusetzen, an der er sie unterbrochen hat, als die Migration begann.

- „an jener Stelle fortzusetzen“ - man beachte die Möglichkeit, mehrere Threads im Agenten zu haben!
- Das heißt, alle Threads müssen wieder an die „richtige Stelle“ platziert werden.

Echte starke Migration muss also immer die Threadmigration beherrschen

Starke Migration

- Charakterisierung (Illmann et al.)



Besondere Bedingungen

- Codemigration
  - Sicherstellen, dass alle Klassen übertragen werden (oder erreichbar sind)
- Codeposition
  - Aufrufstapel (Abfolge der Methodenaufrufe, Call stack) muss ebenfalls gesichert werden
- Threads
  - Neben den Codepositionen in den Threads muss der Threadzustand gesichert werden (schlafend, laufend, blockiert usw.)
  - Thread mit Migrationsaufruf hat vorherbestimmte Codeposition bei Migration; alle anderen Threads nicht!

Besondere Bedingungen

- Felder
  - einfach zu lösen durch Serialisierung / Deserialisierung
- Stack
  - beinhaltet die lokalen Variablenbelegungen aller Methoden auf dem Aufrufstapel
- Ressourcen
  - Externe Ressourcen: Dateien, Sockets, Datenbanken, Fenster,...
  - kann für einige Fälle auf entfernte Kommunikation über Sockets zurückgeführt werden

Schwache Migration

- Alternative zur Wiederherstellung der Codeposition
  - Initialisierungsmigration
    - Agent beginnt seine Ausführung stets in derselben Methode
    - Werte im Zustand dienen zur Auswahl der Codeposition (feste Positionen) nach der Migration
    - ggf. auch in Form von Ereignisbehandlung
    - meistverwendete Variante
  - Methodenmigration
    - Spezifizierung einer Methode, welche nach der Migration ausgeführt wird

Unter einer **schwachen Migration** versteht man eine Migration eines Agenten, die den Agenten an der Zielposition in Bezug auf seinen Zustand und seine Beschreibung als präzise Kopie des Ursprungsagenten erzeugt, jedoch die Ausführung nur an genau vorbestimmten Stellen wieder aufnehmen kann.

Migration

- Beispiel
  - Was würde ein Agent mit dieser Beschreibung tun?

```
public void myMethod() {
        String[] as = { "place1", "place2", "place3" };
        for (int i=0; i < as.length; i++) {
                migrateTo(as[i]);
        }
}
```

Übungsaufgabe!

Migration in Java

- Schwache Migration
  - kein Problem für Java (nur Serialisierung + dynamisches Laden von Klassen)
- Jedoch von großer Relevanz innerhalb der Anwendung
  - Programmierer muss Vorkehrungen treffen, dass der Agent „weiß“, wie er weiterzumachen hat

```
switch(nPosition) {
        case 1: nPosition=2;
                break;
        case 2: nPosition=3;
                break;
...
}
```

nPosition als Teil des Zustands

Agent „spult“ die Methode bei Wiederaufnahme vor

Starke und schwache Migration

- Vorteil der starken Migration
  - Leicht für den Anwender zu verstehen
  - Keine speziellen Vorkehrungen im Code nötig
    - Aber ist das so? Muss man wirklich nichts tun, wenn der Agent migriert ist? (Multiagentenanwendungen!)
- Vorteil der schwachen Migration
  - Direkt von Java unterstützt und leicht zu realisieren
  - Geringer Aufwand im System, effizient

Wie häufig kommen Migrationen vor?
Wie sehr sind sie im Code verstreut?
Lohnt sich der Aufwand für starke Migration?

Simulation starker Migration

- Simulation von starker Migration über schwache Migration
  - Anwenden eines Präprozessors
  - Anwender programmiert im Stile einer starken Migration
  - Quellcode wird modifiziert
  - System führt schwache Migration aus

Konvertierung

```
public int beispiel() {
        int a=10;
        String sResult;
        sResult = methode();
        migrate(...);
        return a;
}
```

```
public int beispiel() {
        int a;
        String sResult;
        if (m_bMigration!=true) {
                a=10;
                sResult = methode();
                StackFrame sf = new StackFrame();
                sf.save(new Integer(a), 1);
                sf.save(sResult, 2);
                state.push(sf);
                m_bMigration=true;
                migrate(...);
        }
        else {
                StackFrame sf = state.pop();
                a = ((Integer)sf.restore(1)).intValue();
                sResult = (String)sf.restore(2);
                m_bMigration = false;
        }
        return a;
}
```

```
public class StackFrame {
        public void save(Object o, int pos);
        public Object restore(int pos);
}
```

```
public class State {
        public void push(StackFrame sf);
        public StackFrame pop();
        ...
}
```

- nach Illmann et al.: Migration of Mobile Agents in Java: Problems, Classification and Solutions, MAMA 2000

Konvertierung

- Sehr komplex bei Programmstrukturen wie Schleifen
  - Initialisierungen separat
  - Schleifenzustand sichern

```
int i=3;
int a=1;
while (i > 0) {
        i--;
        a = a*2;
        int b=a;
        migrate(...);
        a = a+1;
}
```

```
int i;
int a;
if (m_bMigration==false) {
        i=3;
        a=1;
        ... // Speichere i,a
        state.push(frame);
}
else {
        StackFrame frame = state.next(); // behält auf Stack
        ... // Wiederherstellen von i,a
}
while (i > 0) {
        int b;
        if (m_bMigration==false) {
                i--;
                a=a*2;
                b=a;
                ... // Speichere i, a, b
                state.push(frame);
                m_bMigration = true;
                migrate(...);
        }
        else {
                StackFrame frame = state.pop();
                ... // Wiederherstellen von i, a, b
                m_bMigration = false;
                a=a+1;
        }
}
state.pop();
```

Achtung:

int i=3;
  muss umgewandelt werden in
int i; ... i=3;

Modifikation des Bytecodes

- Verwendung des JVMDI (Java VM Debugger Interface), um die Threadrahmen zu bekommen
- Manipulation des Bytecodes, um die Threadposition zu setzen

aufrücken

```
Method void myMethod(java.lang.String[])
    // >>>> BEGIN of inserted code
    1 iconst_0 ; 0 auf Stack
    2 istore 99 ; in Slot 99 ablegen
    3 iload 99 ; auf den Stack
    4 tableswitch 0 to 2: default=29
        0: 29
        1: 55
        2: 69
    // <<<< END of inserted code
    29 ... // Originalcode
    ...
    54 invokevirtual #5 <Method void migrate()>
    ...
    68 invokevirtual #5 <Method void migrate()>
    ...
    75 return ;
```

zwei Migrations-anweisungen

Modifikation des Bytecodes

- Nutzung des Single-Step-Modus (Debugger)
  - ersten zwei Anweisungen werden mittels SS übersprungen
  - richtigen Sprungwert setzen
  - Single-Step beenden und Thread weiterlaufen lassen
- Vorteil
  - keine Änderung am Quellcode
  - echte starke Migration
- Nachteil
  - Erheblicher Performanzverlust durch Debugger

Andere Strategien

- Codeposition und Stacksicherung
  - Modifikation der Java Virtual Machine
  - Beispiel Ara (H. Peine)
- Vorteil
  - keinen Einfluss auf die Programmierung
  - effizient
- Nachteil
  - Modifikation der Java-VM
  - keine Interoperabilität mehr

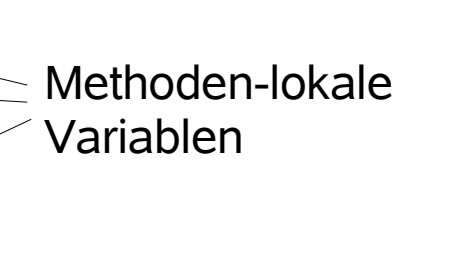
Andere Strategien

- Ansatz
  - Einsatz in WASP (S. Fünfrocken, TU Darmstadt)
  - Präprozessor fügt Zustandssicherung lokaler Variablen in den Quellcode ein
  - Nach Migration Wiederherstellen durch Deserialisierung und durch Auslesen des Zustandsobjekts für lokale Variablen
  - Überspringen von Codeblöcken beim Neustart
    - wie schon früher beschrieben (if-else, switch)

Starke Migration und Präprozessor

- Beispielprogramm

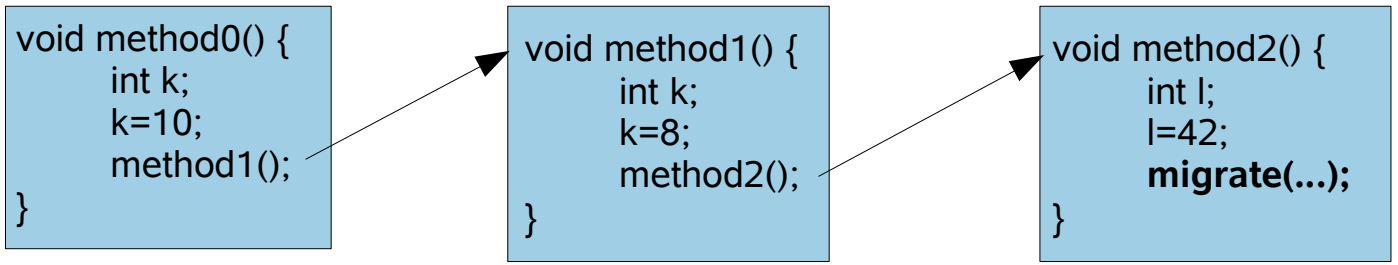
```
class MyProgram {
    ...
    public void mymethod(int i, float j, MyObject m) {
        int k;
        Hashtable h;
        ...
        migrate();
        ...
        if (k==5) {
            Vector x = new Vector();
            ...
            migrate();
            ...
        }
        int v = 10;
        ...
        migrate();
    }
}
```



aus S. Fünfrocken: Transparent Migration of Java-based Mobile Agents (MA 98)

Starke Migration und Präprozessor

- Problem Aufrufketten
  - Jede Methode hat ihre eigenen lokalen Variablen
  - Nur innerhalb der Methode selbst zu sichern
- Performanzproblem
  - Man müsste in einer Aufrufkette vor jedem Aufruf den lokalen Zustand sichern
- Ansatz
  - Zustandssicherung wird nur durch die Methode angestoßen, die die Migration auslöst



Aufrufstack:

```
migrate
method2
method1
method0
```

Starke Migration und Präprozessor

- Sichern der Aufrufkette durch Werfen von

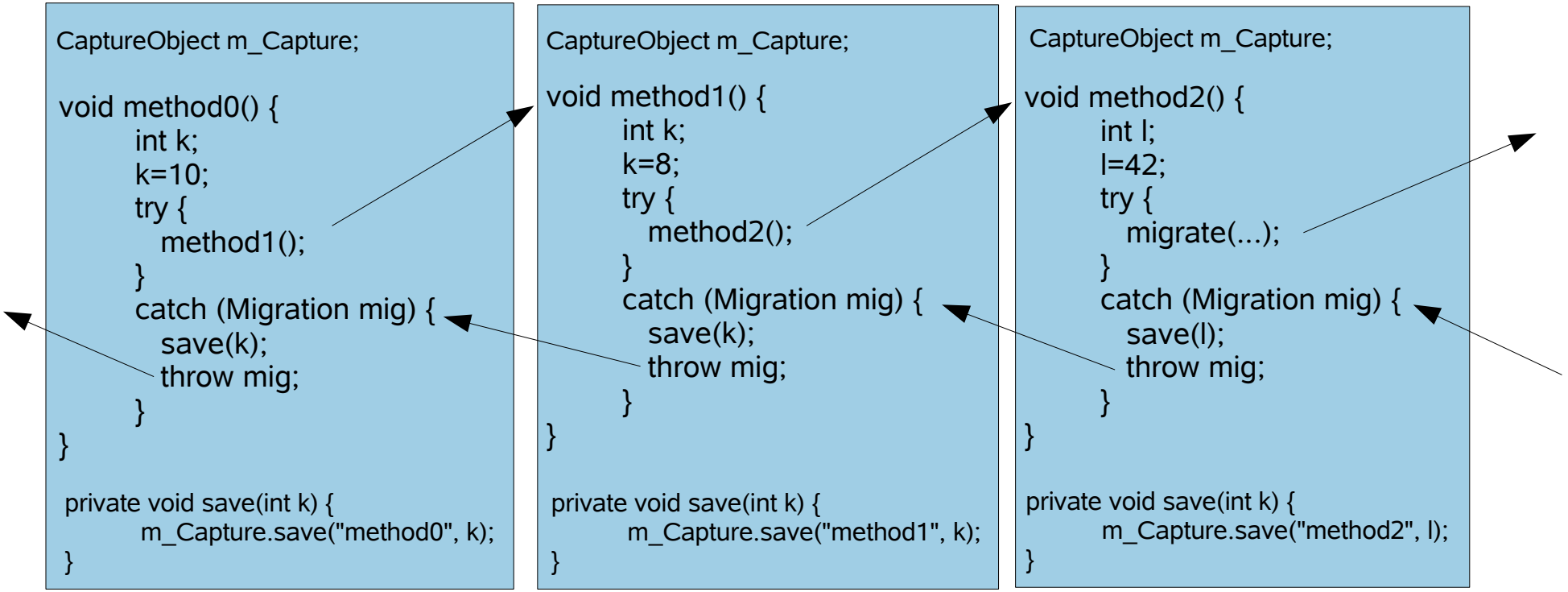
```
public class Migration extends java.lang.Error
```

- Migrationsmethode wirft diesen Error
  - Vorteil: muss nicht deklariert werden in den Methodensignaturen

```
public void migrate(String sDestination) throws Migration;
```

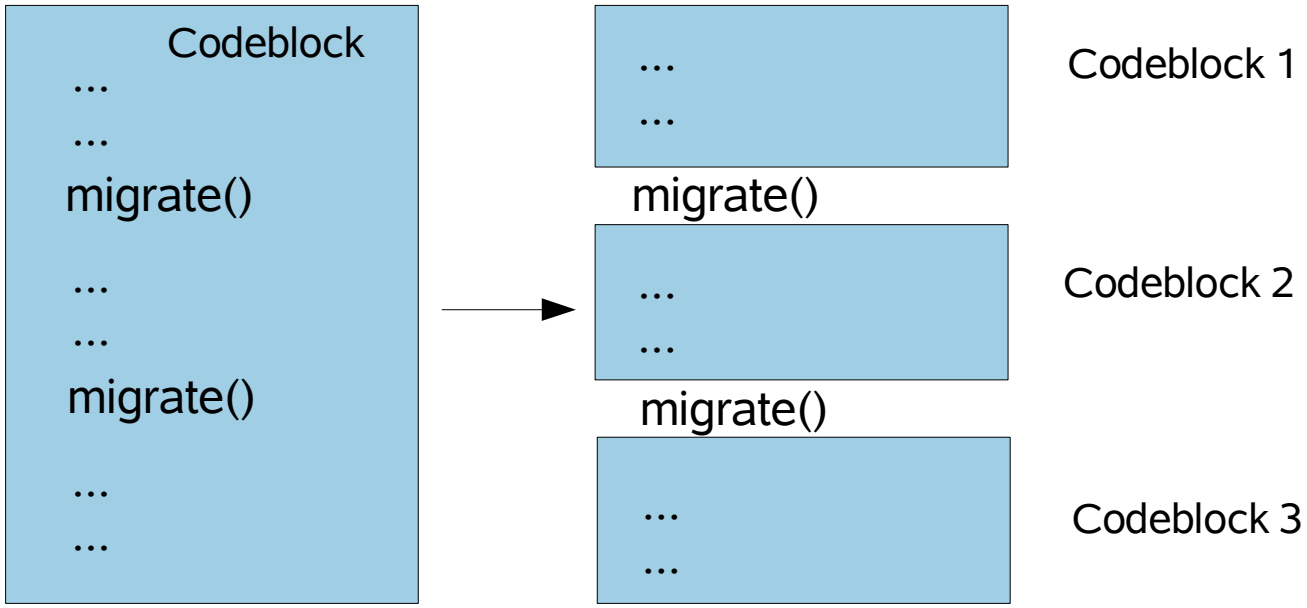
Zustandssicherung

- Zugriff auf Zustandssicherungsobjekt (m_Capture)
  - Alle aggregierten Klassen benötigen Referenz
  - Modifikation des Quellcodes



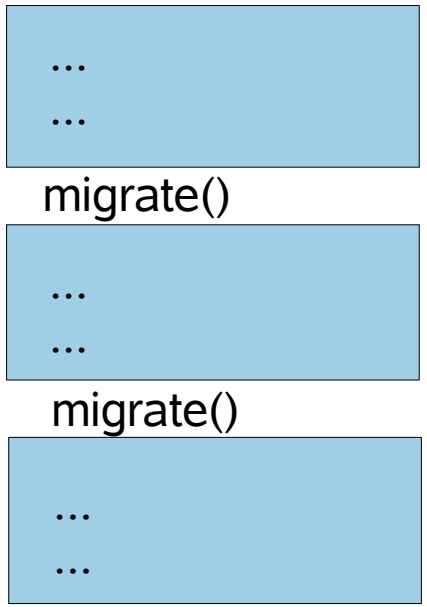
Wiederherstellen der Codepositionen

- Unterteilung des Programms in Blöcken



Wiederherstellen der Codepositionen

- Hinzufügen von if-Strukturen
  - entsprechende Problematik bei Verzweigungen, Schleifen




```
if (block==1) {
        ...
        ...
        block=2; migrate();
}
if (block==2) {
        ...
        ...
        block=3; migrate();
}
if (block==3) {
        ...
        ...
}
```

Threadsynchronisation

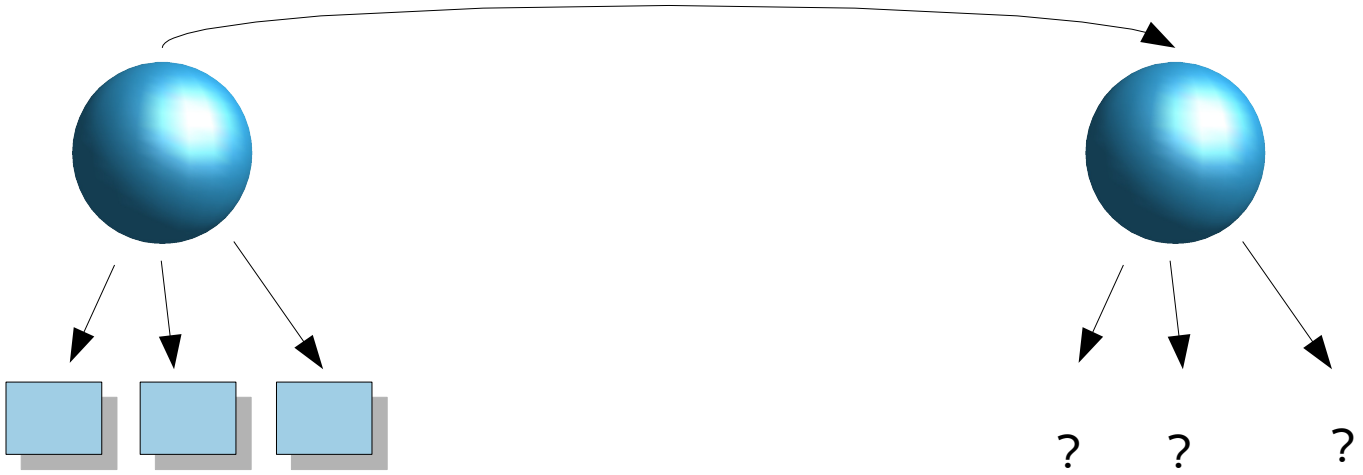
- Multithread-Agenten
  - Mehrere Threads, aber nur ein Thread ruft **migrate** auf
  - Wo sind die anderen Threads??

Die Annahme, dass die Codeposition bei der Migration wohlbestimmt ist, ist ohne weiteres nicht gültig bei Multithread-Agenten.

- Problem: nicht automatisch zu lösen
- Ausweg
  - Programmierer muss Unterstützung leisten
  - Methode **allowGo** dient zur Synchronisation der Threads: Migration ist nur möglich, wenn jeder Thread **allowGo** aufgerufen hat

Grenzen der Ansätze

- Ressourcen: Großes Problem!
  - Ressourcen sind an der neuen Position nicht mehr vorhanden! (Dateien, Sockets, ...)
  - Ggf. zu behandeln über entsprechende verteilte Infrastruktur (Öffnen einer Datei hier, Lesen von woanders)



Zusammenfassung

- Kommunikation
  - Zahlreiche mögliche Ansätze, aber welcher ist geeignet für umhermigrierende Agenten?
  - Methodenaufruf sehr bequem, aber unflexibel; behindert ggf. Autonomie
  - Nachrichtenversand sehr elementar, dafür aber flexibel
- Migration
  - Starke und schwache Migration
  - Starke Migration (scheinbar) einfacher für die Agentenentwicklung
    - Jedoch erhebliche technische Anforderungen
  - Ist starke Migration überhaupt notwendig?