CORBA
Lebensdauer von Objekten, Transaktionen

Lebensdauer von Objekten in einem Programm

- ...wird häufig von Programmiersprachen durch Sichtbarkeitsregeln gesteuert, z.B. in Java:

```
void example() {
...
   {  String name = new String("Egon");
       ...
     // am Ende dieses Blocks wird das Objekt
   } // freigegeben

   String s = name; // Compiler-Fehler
}
```

Lebensdauer von Objekten in einem Programm

- in C++

```
void example(int x)
{
  tollesObjekt obj; // zur Laufzeit wird
                    // automatisch ein Objekt
                    // erzeugt
  obj.my_method(x);
} // wenn auf obj nicht mehr zugegriffen werden
  // kann, wird das Objekt automatisch gelöscht
```

Lebensdauer von Objekten in einem Programm

- in C++ ist der Programmierer für die Lebensdauer selbst angelegter Objekte selbst verantwortlich

```
void example(int x) {
   tollesObjekt * obj = new tollesObjekt(x);
   obj->sayHello();
}
```

- Objekt bleibt im Speicher, der Zeiger ist aber nicht mehr zugänglich
- so entstehen Memory Leaks

Lebensdauer von Objekten in einem Programm

```
void example(int x) {  // noch ein C++ Beispiel
   tollesObjekt * obj = new tollesObjekt(x);
   obj->sayHello();
   delete obj;
   obj->sayHello(); // FEHLER! Objekt schon gelöscht
}
```

- Zeigern sieht man nicht an, ob sie auf gültige Objekte zeigen oder nicht - große Fehlerquelle
- in Java gibt es deshalb keine Zeiger sondern Objekt-Referenzen, die intern mitzählen, wieviele Verweise es auf ein Objekt gibt - wenn der Referenz-Zähler Null enthält wird das Objekt gelöscht (es gibt kein delete)

Lebensdauer von Objekten in einem Programm

- unabhängig von der Programmiersprache: ein Java- oder C++-Objekt existiert nur im Speicher und ist nach der Beendigung des Programms verschwunden
- gleiches gilt für Objekt-Referenzen oder Zeiger
- da Programme im allgemeinen nur kurz laufen (Maximum: die Zeit zwischen zwei Stromausfällen), werden Programmiersprachen-Objekte auch als "transient objects" bezeichnet (Objekte kurzer Lebensdauer)
- Häufig soll der Zustand eines Objektes jedoch länger erhalten bleiben, z.B. durch Speichern des Zustandes auf der Festplatte oder in einer Datenbank - die Objekte werden dadurch zu "persistent objects"

Java-Beispiel: Persistenz durch Serialisierung

- Der Zustand eines Objektes kann z.B. in eine Datei ausgegeben werden, wenn alle Zustandsvariablen dieses unterstützen (wenn sie java.io.Serializable implementieren)

```
import java.io.*;
class Example implements Serializable
{
   String importantState; // implements Serializable
   transient String unimportantState;
} ...
   ObjectOutputStream p = new ObjectOutputStream(...);
   p.writeObject(obj);
...
```

Persistente Objekte in verteilten Systemen

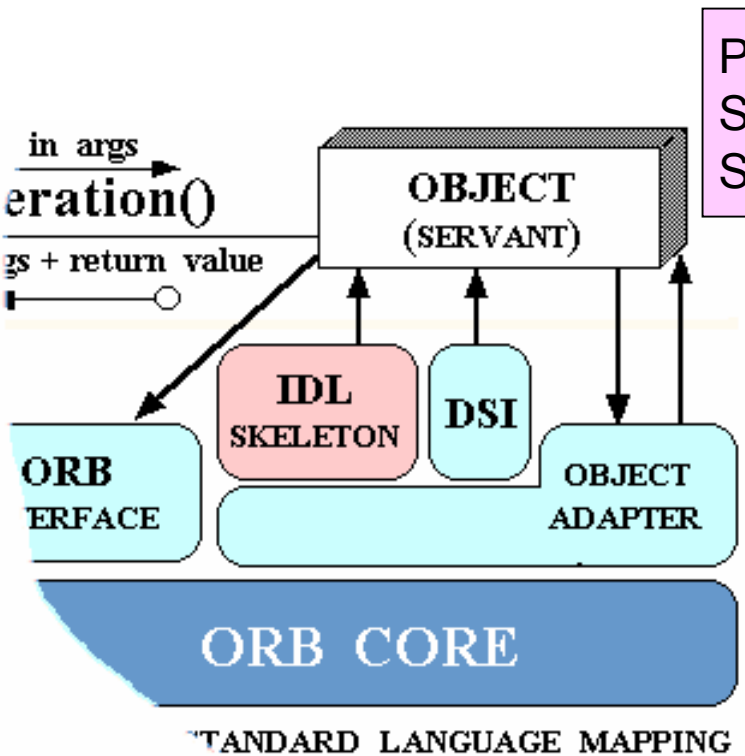
- in verteilten System haben die beteiligten Programme normalerweise unterschiedliche unvorhersagbare Laufzeiten
- der Aufenthaltsort eines Objektzustandes (Speicherbereich, Rechner) kann sich ändern
- Referenzen von verteilten persistenten Objekte dürfen sich aber nicht ändern - sie dürfen keine kurzlebigen Informationen (z.B. Zeiger auf Speicherbereiche eines Rechners) enthalten
- mögliche Lösung: Namensdienst, bei dem unter einem unveränderlichen Namen die aktuelle kurzlebige Referenz gespeichert werden kann
- oder: unveränderliche Referenz (Beispiel: random.org)

Persistente Objekte in CORBA

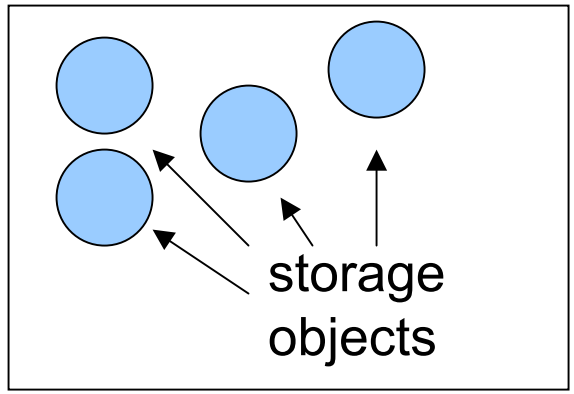
- ORB und POA ermöglichen die Erstellung und Verwaltung von unveränderlichen Objekt-Referenzen
- für die Speicherung von Objekt-Zuständen kann man CORBA-PSS (Persistent State Service) nutzen
  - dient beim Server als Verbindungsglied zwischen Objekt-Servant und einer Datenbank (bzw. einer Datei)
  - automatische Code-Generierung aus einer "Persistant State Description Language (PSDL)"-Datei (ähnlich IDL)
  - kann Datenbank-Tabellen oder Dateien verwalten
  - ungeeignet für die Anbindung vorhandener Datenbank-Tabellen
  - die Konzepte des PSS lassen sich aber gut in eigenen Programmen verwenden

Persistent State Service (PSS)

Konzept - stark vereinfacht



PSDL
Stubs &
Skeletons



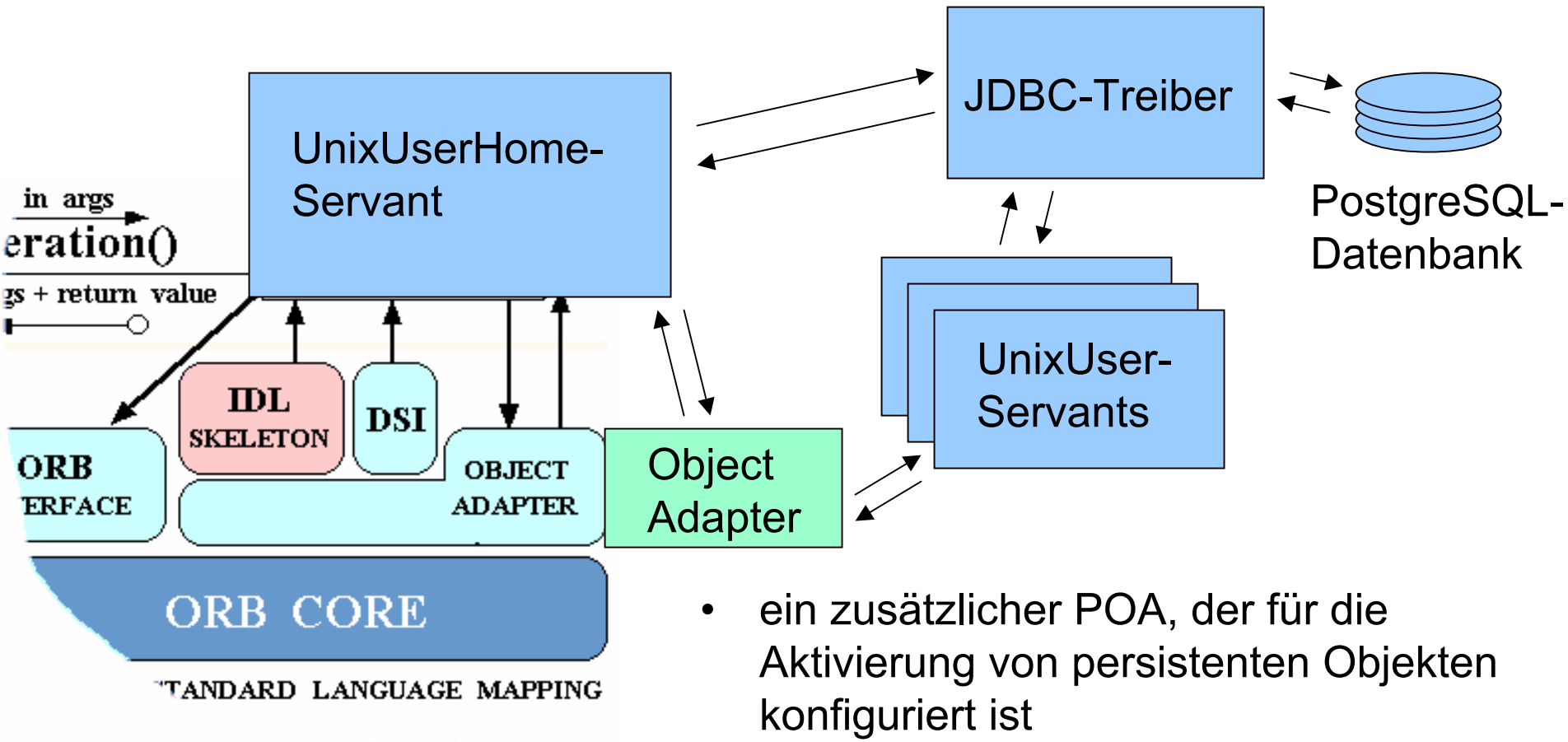
storage home

- PSS ist ein Service, der lokal im Server-Prozess verwendet wird - keine CORBA-Objekte
- Idee: "storage objects" haben ein Zuhause
- "storage home"-Objekte wissen, wie man "storage objects" findet

Beispiel: Benutzerdatenbank - persistente CORBA-Objekte

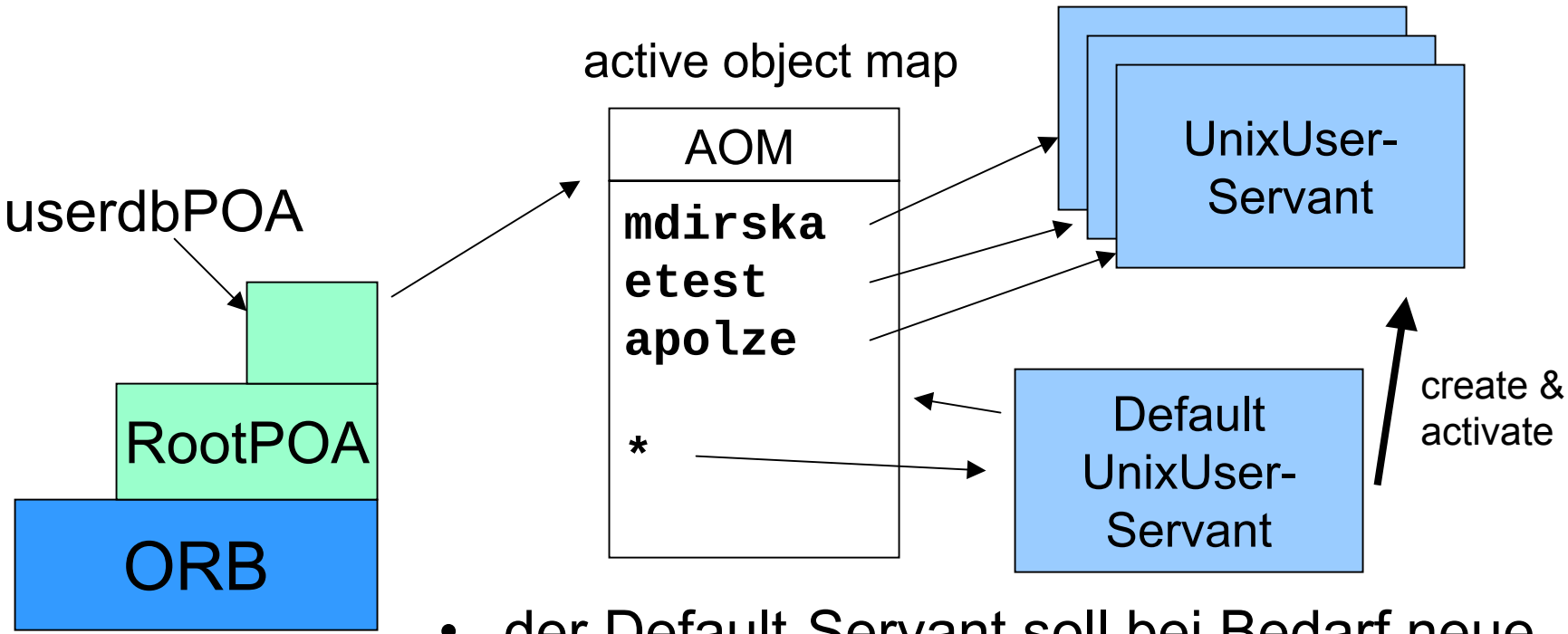
```
// IDL
...
interface UnixUser  // Zustand ist in Datenbank gespeichert
{
        readonly attribute string sid; // eindeutiger String
        readonly attribute string login;
        readonly attribute long   uid;
        readonly attribute long   gid;
        readonly attribute string name;
        readonly attribute string homedir;
        attribute UnixUserStatus status; // NEW, READY, ...
};
...
interface UnixUserHome
{
        UnixUser find_by_login (in string login);
};
```

Beispiel: Benutzerdatenbank



- ein zusätzlicher POA, der für die Aktivierung von persistenten Objekten konfiguriert ist

zusätzlicher Portable Object Adapter: userdbPOA

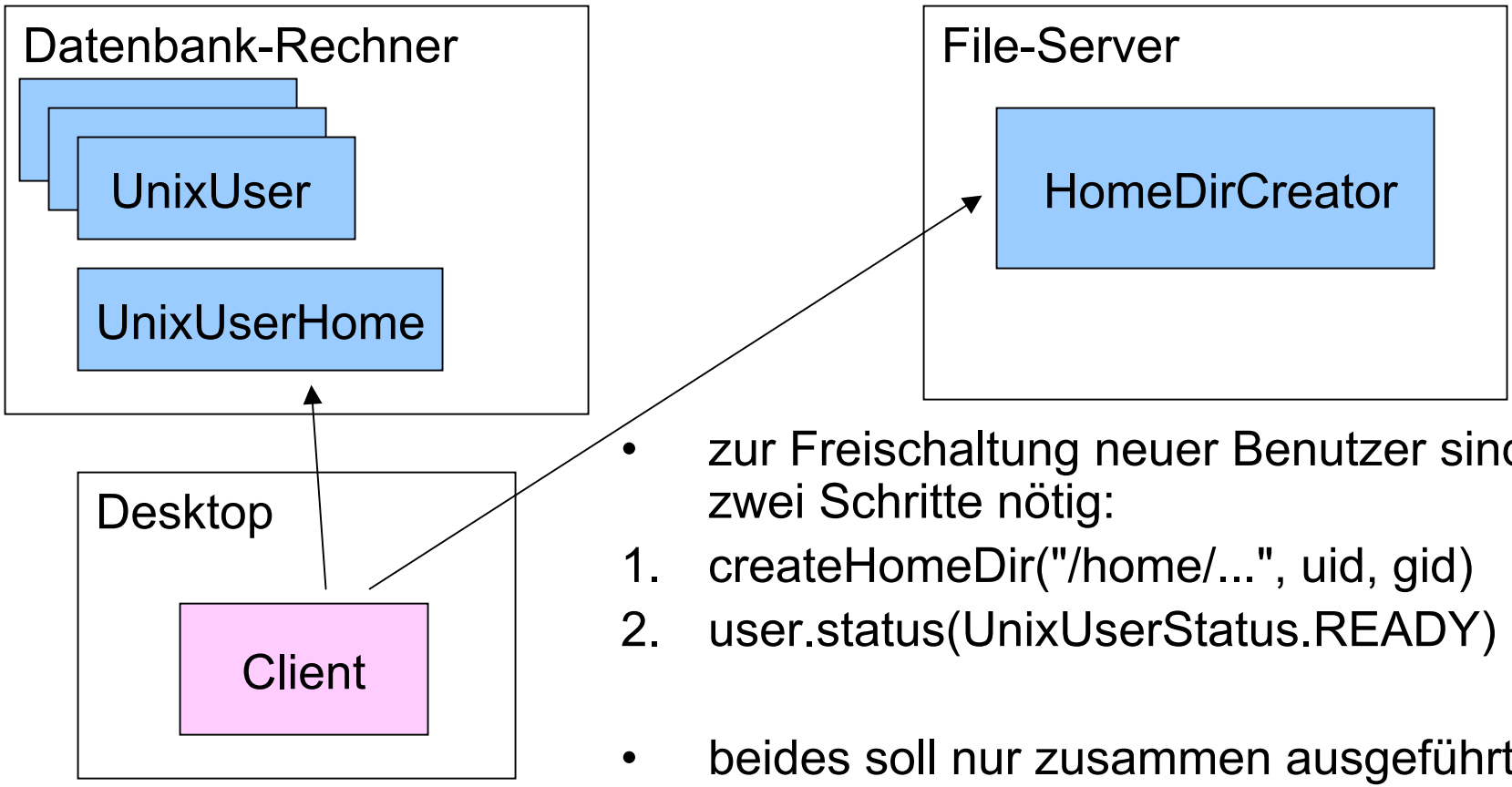


- der Default-Servant soll bei Bedarf neue Servants erzeugen und aktivieren

POA Policies

Policy Values	default	RootPOA	userdbPOA
PERSISTENT			X
TRANSIENT	X	X	
SYSTEM_ID	X	X	
USER_ID			X
NO_IMPLICIT_ACTIVATION	X		X
IMPLICIT_ACTIVATION		X	
RETAIN	X	X	X
NON_RETAIN			
USE_ACTIVE_OBJECT_MAP_ONLY	X	X	
USE_SERVANT_MANAGER			
USE_DEFAULT_SERVANT			X
UNIQUE_ID	X	X	
MULTIPLE_ID			X
SINGLE_THREAD_MODEL			
ORB_CTRL_MODEL	X	X	X

Anwendung: Freischaltung von Benutzern



- zur Freischaltung neuer Benutzer sind zwei Schritte nötig:

1. createHomeDir("/home/...", uid, gid)
2. user.status(UnixUserStatus.READY)

- beides soll nur zusammen ausgeführt werden - "ganz oder gar nicht"

Verteilte Transaktionen mit CORBA

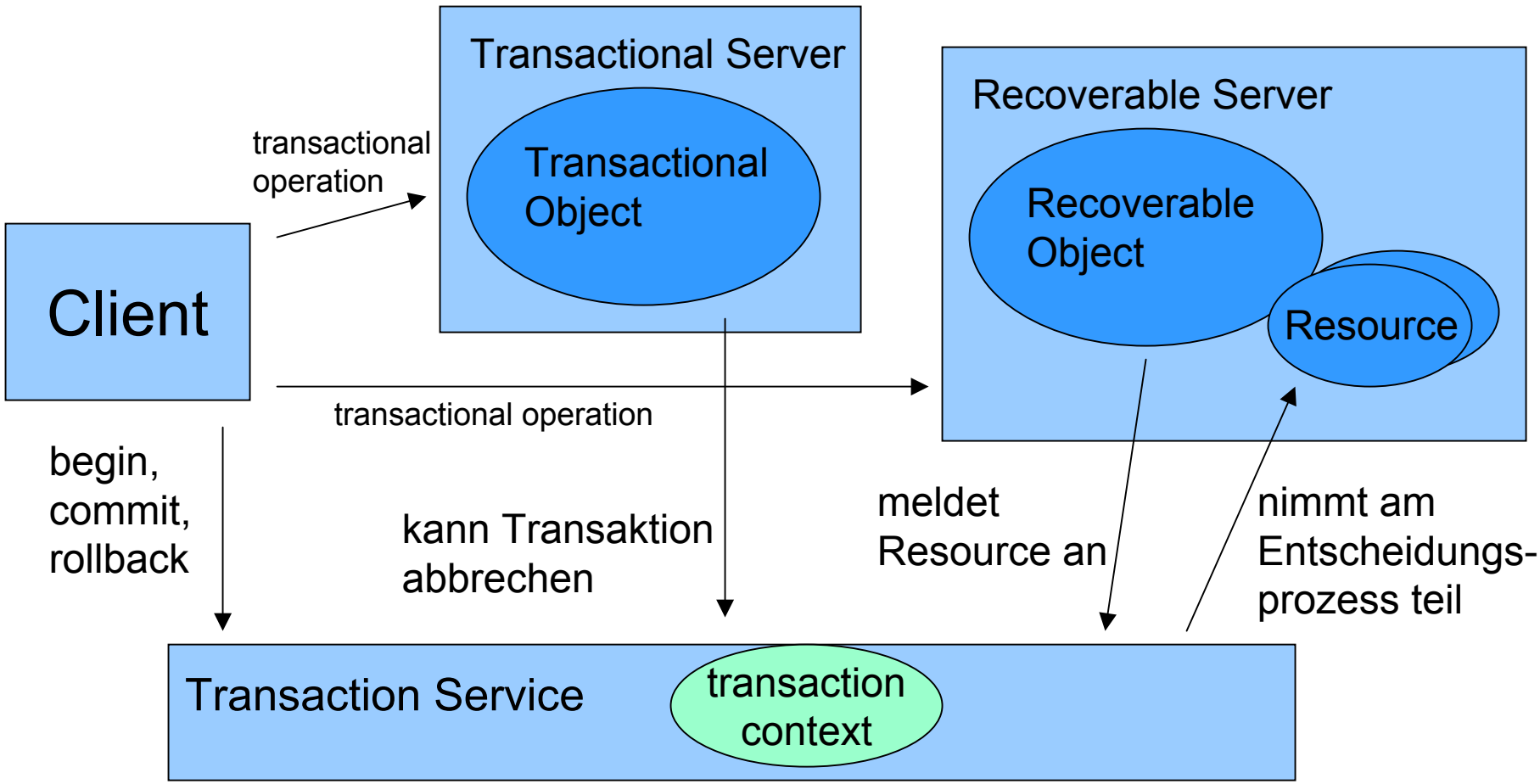
- CORBAservices - Transaction Service
- Transaktionen sollten folgende Eigenschaften haben (ACID):
  - **A**tomic - sie erscheinen nach außen wie *eine* Operation; wenn eine Transaktion abgebrochen werden muss wird der ursprüngliche Zustand wiederhergestellt (rolled back)
  - **C**onsistent - Transaktionen halten das System in einem konsistenten Zustand, d.h., fundamentale Regeln des Systems werden nicht verletzt
  - **I**solated - möglicherweise inkonsistente Zwischenzustände, die während der Ausführung einer Transaktion auftreten, sind nach außen nicht sichtbar
  - **D**urable - der Ergebnis-Zustand einer Transaktion kann und sollte dauerhaft bestehen bleiben, denn es handelt sich um einen gültigen konsistenten Zustand

Das Client-Programm

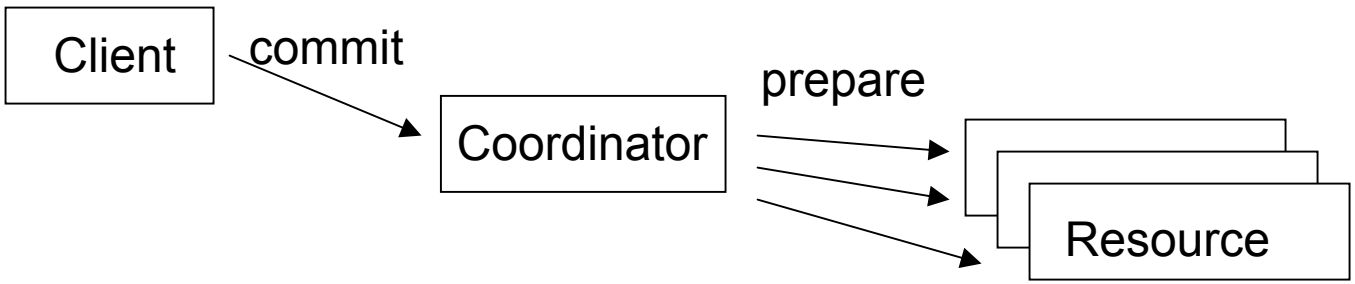
```
try {
 CORBA.Object obj =
   orb.resolve_initial_references("TransactionCurrent");
 txn_current = CosTransactions.CurrentHelper.narrow( obj );

 txn_current.begin();
  hdc.createHomeDir(homedirpath, uid, gid);
  user.status(UnixUserStatus.READY);
 txn_current.commit(false);
}
catch ( org.omg.CORBA.TRANSACTION_ROLLEDBACK ex )
{
   System.out.println("Transaction rolled back !");
}
```

Transaction Service - Bestandteile



two-phase commit protocol (2PC) - Phase 1

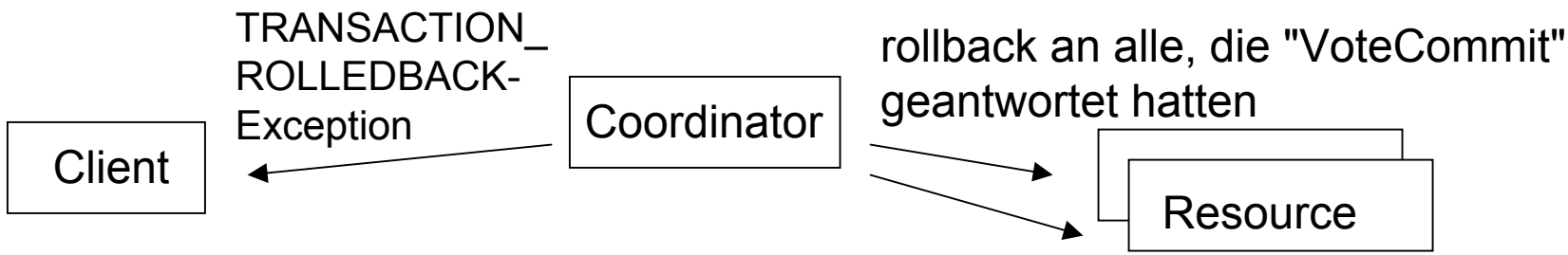


- wenn der Client die Transaktion ausführen möchte (commit) schickt der Transaktions-Koordinator zunächst an alle Resource-Objekte eine prepare-Nachricht
- wenn nur eines der Objekte "VoteRollback" antwortet muss die Transaktion abgebrochen werden
- Resources, die "VoteReadOnly" geantwortet haben, werden nicht weiter berücksichtigt

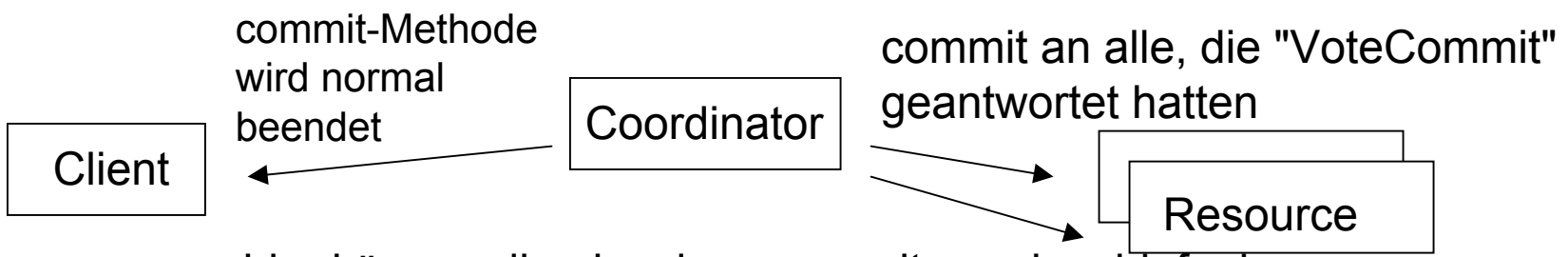
VoteReadOnly
VoteRollback
VoteCommit

two-phase commit protocol (2PC) - Phase 2

- eine Resource hat mit "VoteRollback" geantwortet:



- keine Resource hat mit "VoteRollback" geantwortet:



  - hier können die einzelnen commits noch schiefgehen
  - Heuristic-Exceptions geben Auskunft über den Fehler
  - Client kann beim Koordinator Infos über solche Fehler erfragen