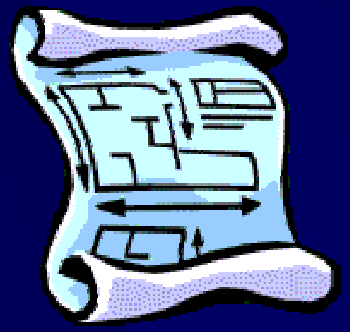
CORBA 3

Michael Stal
Siemens AG, Dept. CT SE 2
E-Mail:
Michael.Stal@mchp.siemens.de

Agenda

- Motivation
- Architectural View
- OMG Organization



- CORBA ORBs and Services
- CORBA 3
  - Internet
  - Quality of Service
  - CORBA Components
  - Portable Interceptor Framework
- Design Issues
- Platform Comparison
- Summary
- References

SIEMENS CORBA

OOP 2001 MÜNCHEN

Building distributed applications is complex

- How to cope with heterogeneity?
- How to access remote services in a location-transparent way?
- How to handle (de-)marshaling issues?
- How to find remote objects?
- How to activate remote objects?
- How to keep state persistent and consistent?
- How to solve security issues?
- Synchronous/asynchronous communication?

SIEMENS CORBA

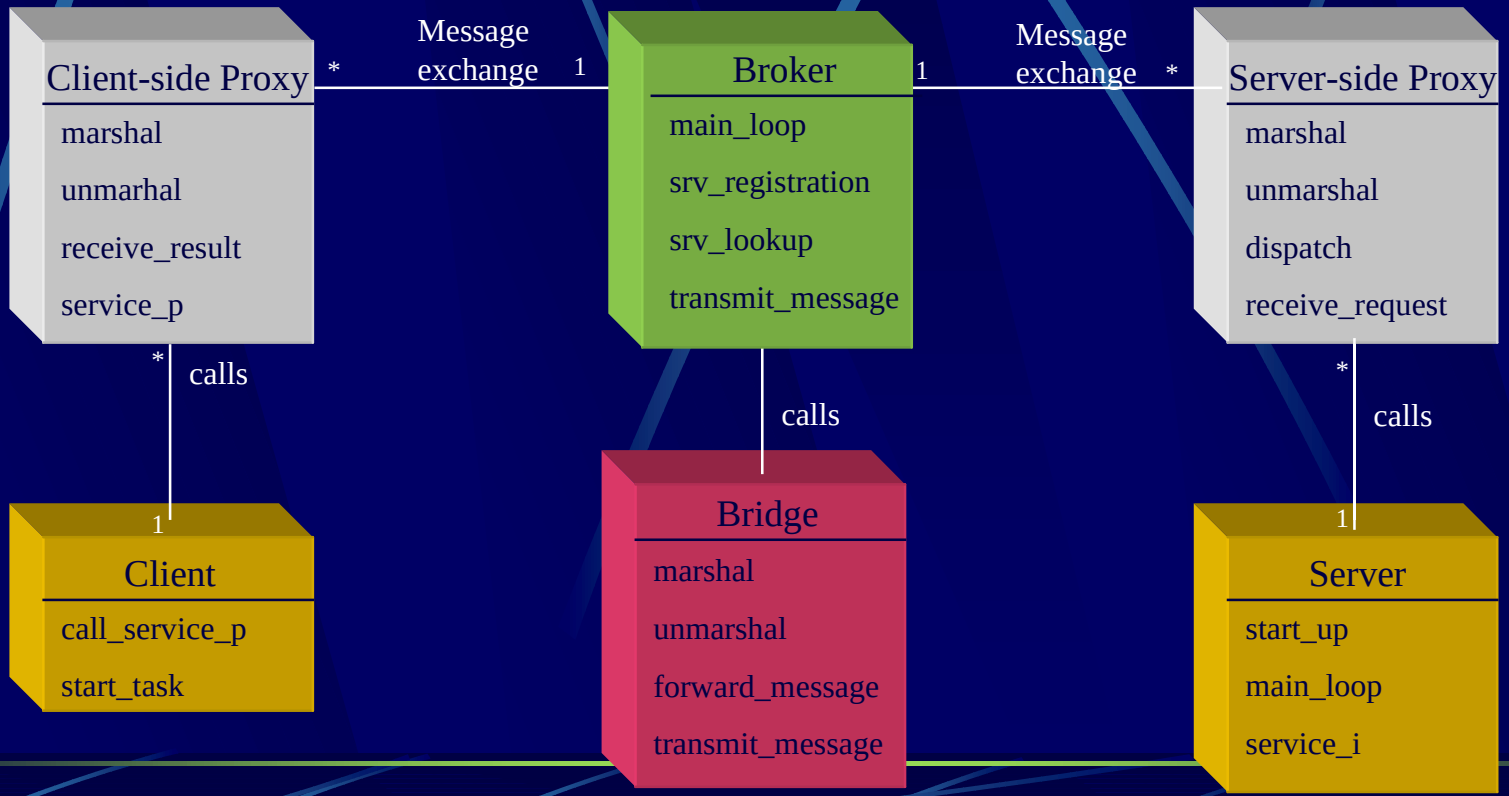
OOP 2001 MÜNCHEN

Distributed Objects are the answer

- What we need is an architecture that ...
  - supports a remote method invocation paradigm
  - provides location transparency
  - allows to add, exchange, or remove services dynamically
  - hides system details from the developer

Architectural Solution

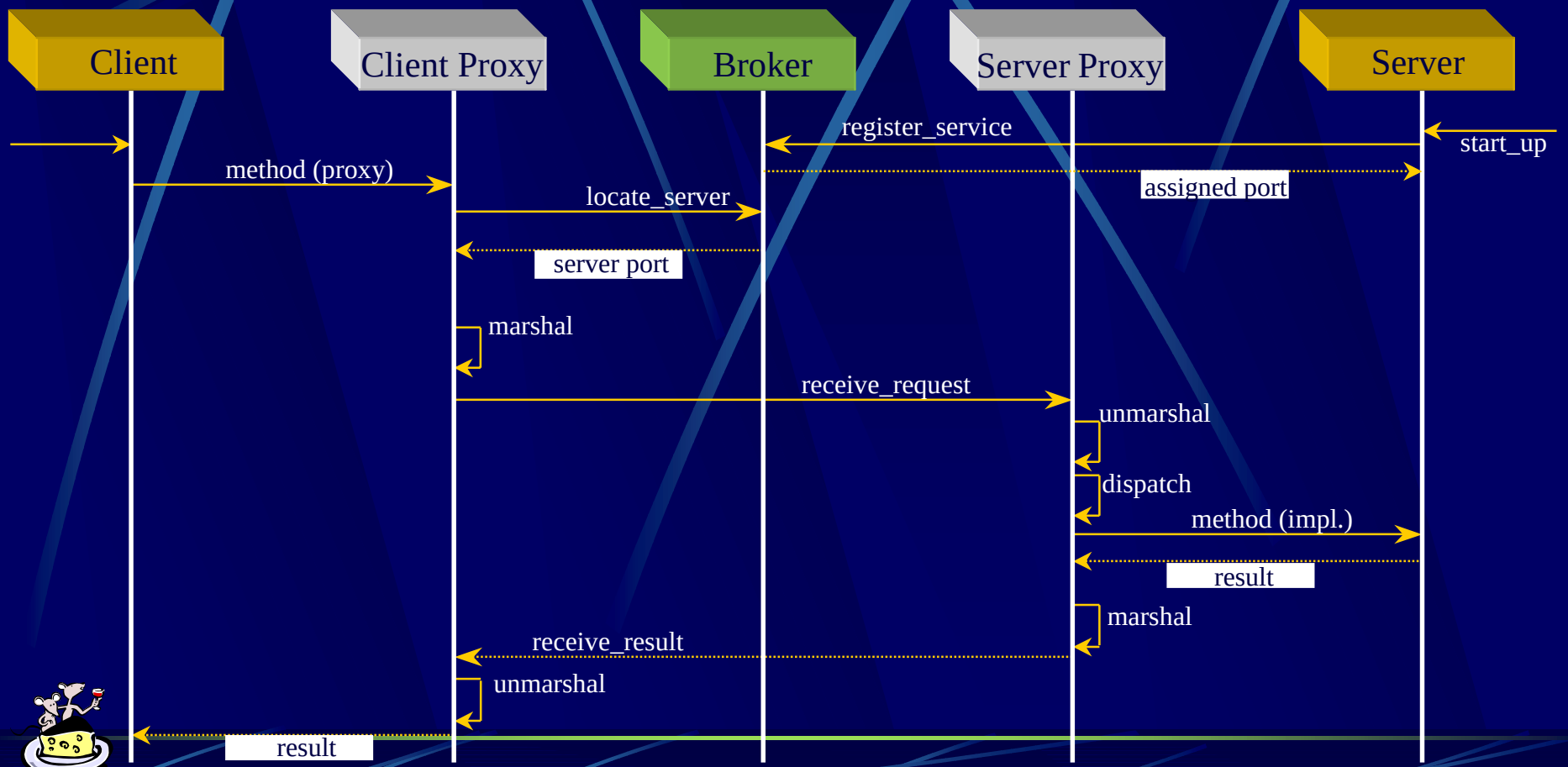
- Here is the architectural solution:



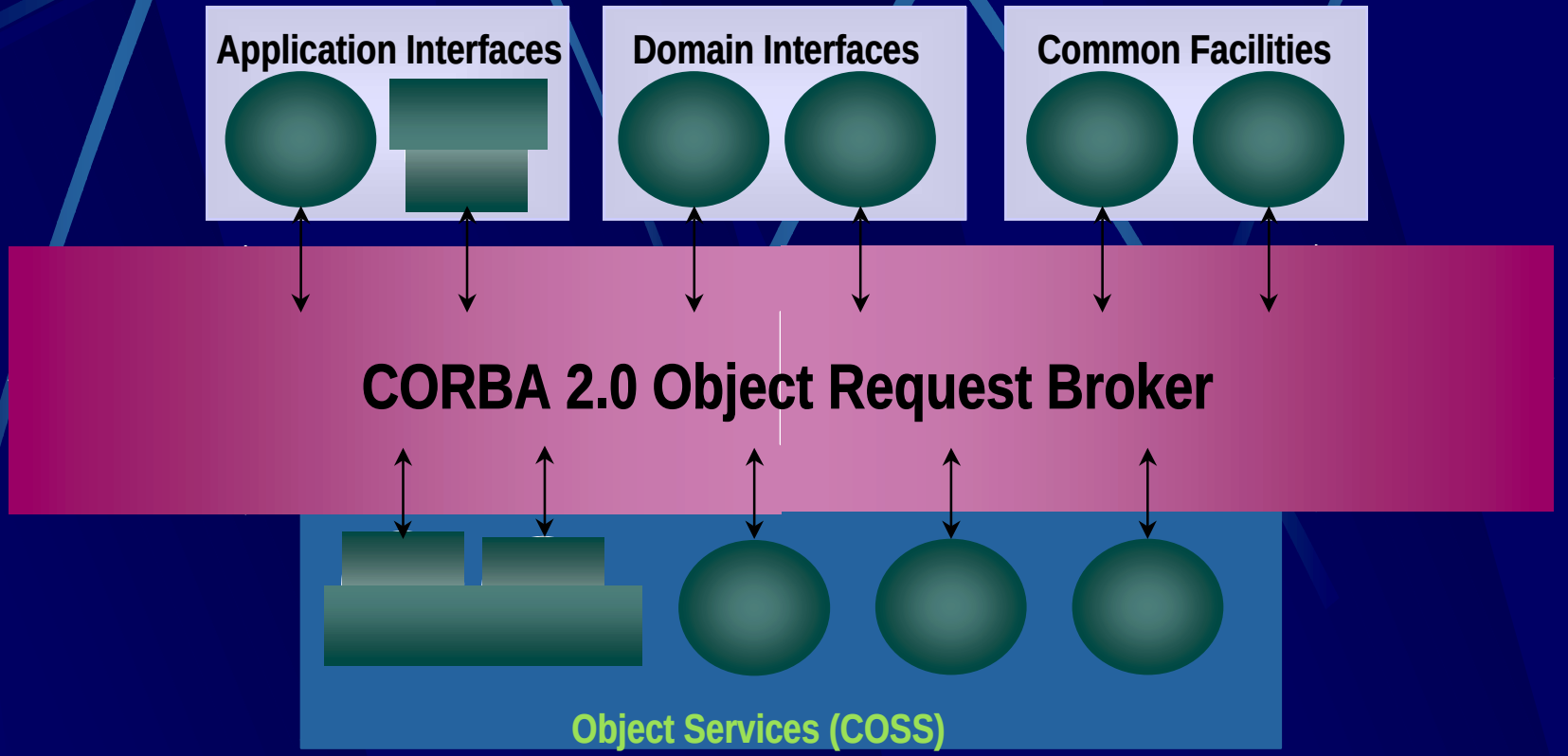
SIEMENS CORBA

OOP 2001 MÜNCHEN

Dynamics of Broker-based systems



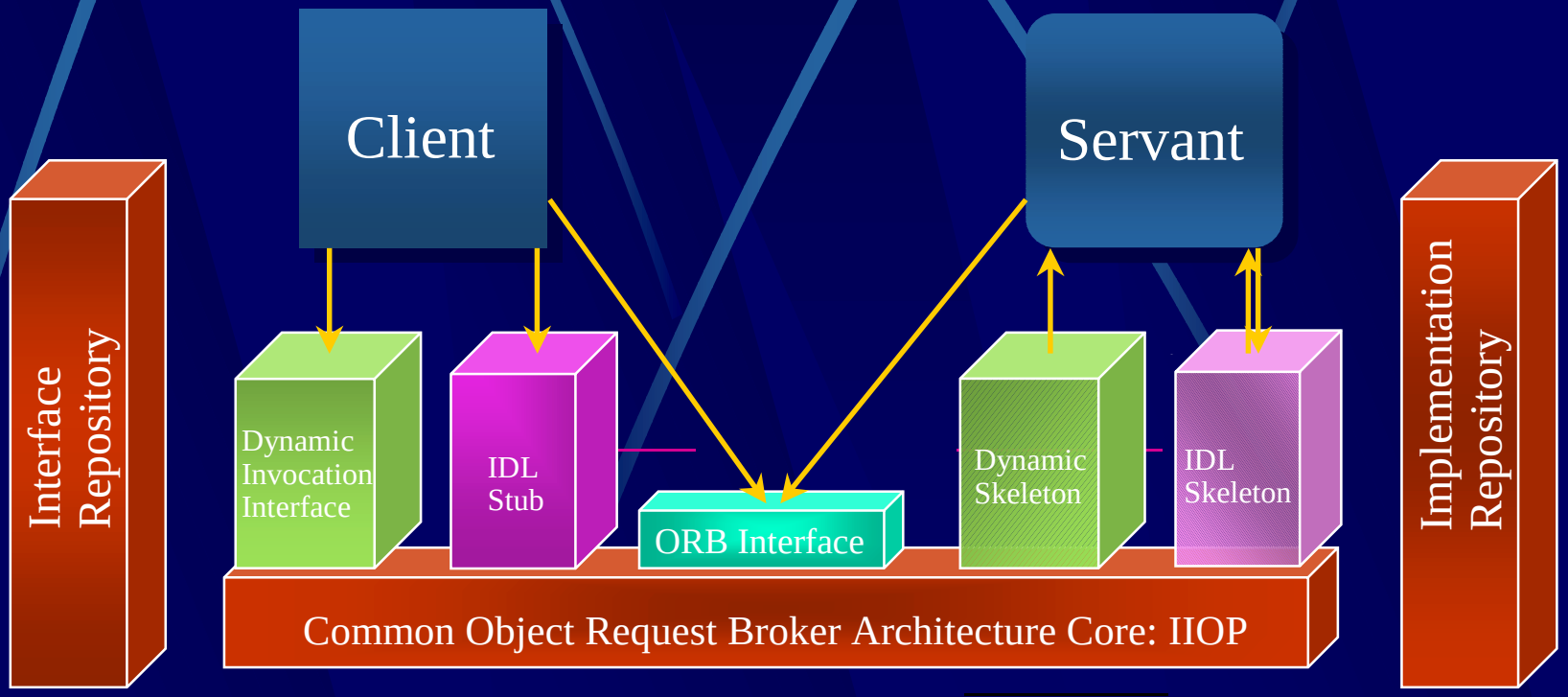
OMG Reference Architecture



SIEMENS CORBA

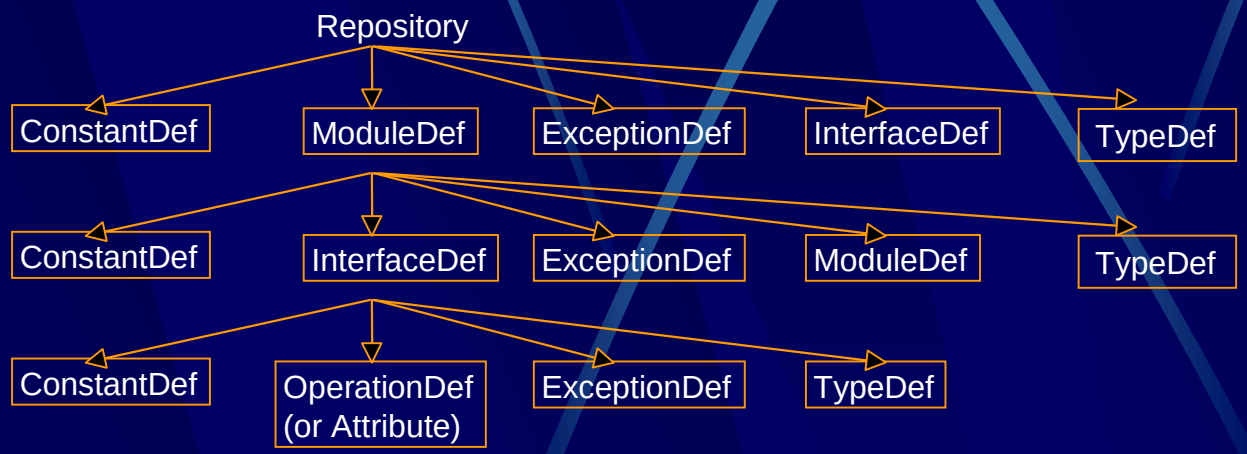
OOP 2001 MÜNCHEN

CORBA Architecture



Interface Repository

The interface repository service is defined as a set of objects specified in IDL:



Implementation Repository

The Implementation Repoitory used for configuration information and for dynamic server activation

Logical Name	Object Adapter	Command	Host	Additional Informat.
Bank	POA("MyPOA")	c:/winnt/test/bank.exe	lotus.muc.bank.de Port: 1234h	SHARED
Account	POA("XPOA");	d:\account.exe	lotus.muc.bank.de Port: 1340h	SHARED
ATM	POA("RemB");		main.fra.bank.de Port: 1340h	SHARED

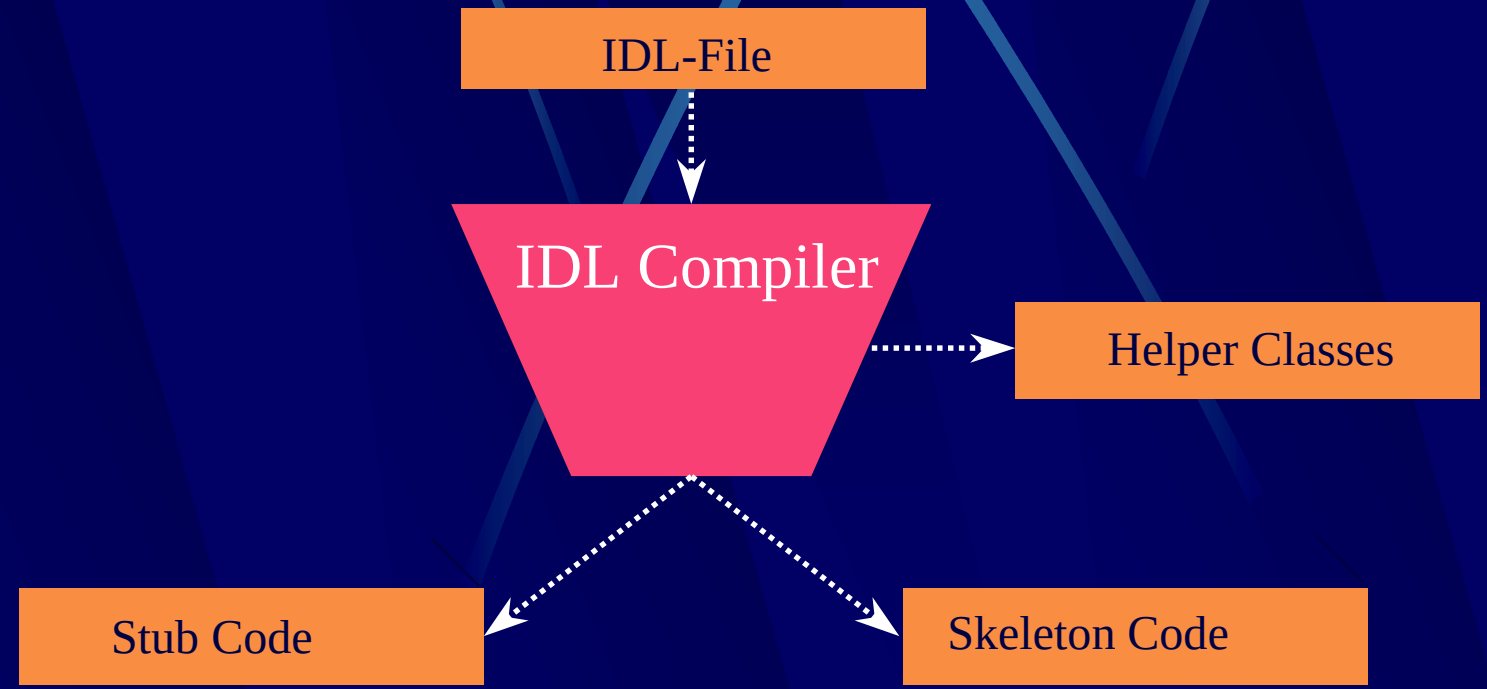
SIEMENS CORBA

OOP 2001 MÜNCHEN

CORBA IDL

- Interfaces are specified in the *Interface Definition Language* IDL. IDL is programming-language independent and does only contain data descriptions.
- An IDL-Compiler translates the IDL-specifications into *IDL-stubs* (for callers) and *IDL-skeletons* (for object implementations) in the appropriate programming language.
- The CORBA Object Model supports multiple *Interface* inheritance. *Attributes* and *Methods* may not be redefined, implementation inheritance is not supported !

CORBA IDL (cont‘d)



SIEMENS CORBA

OOP 2001 MÜNCHEN

How to build and deploy a CORBA application

1. Specify the server interfaces using CORBA IDL
2. Generate stubs and skeletons using the IDL compiler.
3. Implement the server classes.
4. Implement the main routine of the server.
5. Compile server and register it with ORB.
6. If different ORB or different language use IDL compiler.
7. Create client and compile it.
8. Run client.

Example: A Remote Hashtable

IDL File:

```
typedef unsigned long Cookie;
interface Iterator {
   exception InvalidAccess{};
   void reset();
   void skip(in unsigned long n);
   void next();
   any current() raises(InvalidAccess);
   boolean end_of();
};
interface HashTable {
   exception Unknown { string reason; };
   Iterator searchKey(in string key);
   void removeKey(in string key) raises(Unknown);
   void removeEntry(in string key, in Cookie c) raises(Unknown);
   Cookie insertEntry(in string key, in any value);
};
```

Example (cont'd)

Server Header for implementing HashTable:

```
#include "hashS.h"
#include "helper.h"
class Hash_i : public virtual POA_HashTable {
  KeyTable table_;
  Cookie c_;
  CORBA::ORB_ptr orb_;
public:
  Hash_i() : c_(0) {}
  void orb(CORBA::ORB_ptr o) { this->orb_ = CORBA::ORB::_duplicate(o); }
  virtual void shutdown(CORBA::Environment &) { this->orb_->shutdown(); }
  virtual Iterator_ptr searchKey (const char * key);
  virtual void removeKey (const char * key);
  virtual void removeEntry (const char * key, Cookie c);
  virtual Cookie insertEntry (const char * key,const CORBA::Any & value);
};
```

Example (cont'd)

Server C++ file implementing HashTable:

```
Iterator_ptr Hash_i::searchKey (const char * key) {
   KeyTable::iterator iter = table_.find(key);
   if (iter == table_.end())
        return Iterator::_nil();
   Iterator_i *it = new Iterator_i(*((*iter).second));
   Iterator_ptr it_ptr = (*it)._this();
   return it_ptr;
}

void Hash_i::removeKey (const char * key) {
   KeyTable::iterator iter = table_.find(key);
   if (iter == table_.end()) {
        throw new HashTable::Unknown(CORBA::string_dup("key not
   found"));
   } ....
```

Example (cont'd)

Simplified main program of server:

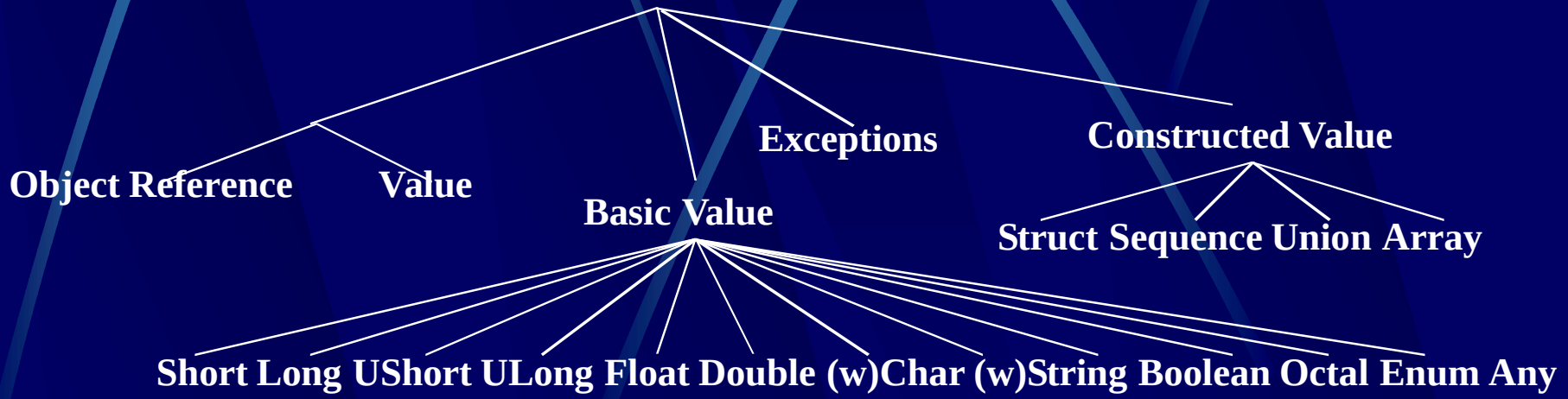
```
int main(int argc, char **argv) {
    MyServer server;
    try {
          if (server.init("test2", argc, argv) == -1)
                return 1;
          else
                server.run(CORBA::Environment::default_environment());
    }
    catch (CORBA::SystemException sysex) {
          ...      }
    catch (CORBA::UserException) {
          ...      }
    return 0;
}
```

Example (cont'd)

Simplified client main:

```
int main(int argc, char **argv) {

    MyClient client;
    if (client.init("test2", argc, argv) == -1)
         return -1;
    cout << "Inserting ... " << endl;
    CORBA::Any a;
    a <<= "http://www.siemens.com";
    Cookie c1 = client->insertEntry("Siemens", a);
    a <<= "http://www.siemens.de";
    Cookie c2 = client->insertEntry("Siemens", a);


    cout << "Iterating ... " << endl;
    Iterator_var iter = client->searchKey("Siemens");
    print_all("siemens", iter); .......
```

CORBA Types

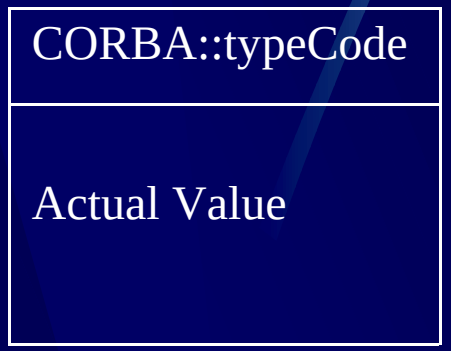


CORBA Types (cont'd)

Type	Range	Size
short	-2^{15} to $2^{15}-1$	>= 16 bits
long	-2^{31} to $2^{31}-1$	>= 32 bits
unsigned short	0 to $2^{16}-1$	>= 16 bits
unsigned long	0 to $2^{32}-1$	>= 32 bits
float	IEEE Single Precision	>= 32 bits
double	IEEE Double Precision	>= 64 bits
char	ISO Latin 1	>= 8 bits
string	ISO Latin 1, w.o. NUL	Variable-length
boolean	TRUE, FALSE	Unspecified
octet	0-255	>= 8 bits
any	Run-time type	Variable-length

Any

- For implementing generic services we need a generic type.
- Example: a generic hash table where all kinds of entries can be stored.
- This is where Any is used.
- Overuse of Any is considered harmful due to potential efficiency problems.
- DynAny even allows to build generic values on the fly.




```
CORBA::Any a;
a <<= (CORBA::UShort) 42;
CORBA::UShort inAny;
a >>= inAny;
```

Structures and Unions

```
union Result switch(ROLE) {
   case ADMIN: string theInfo;
   default: unsigned long theError;
};
union OptionalValue switch(boolean) {
   case TRUE: unsigned short theValue;
};
struct Person {
   string name;
   unsigned long age
};
struct LoveAffair {
   Person p1;
   Person p2;
};
```

Arrays and Sequences

```
typedef string text[20][80];
typedef sequence<Person> Persons;
typedef sequence<Person, 11> SoccerTeam;
struct Tree {
        sequence<Tree> children;
};
```

- Use arrays for fixed length structures where the values lifecycle is coupled with that of the array (Whole-Part relationship).
- Use sequences for recursive types, sparse arrays and variable length structures.

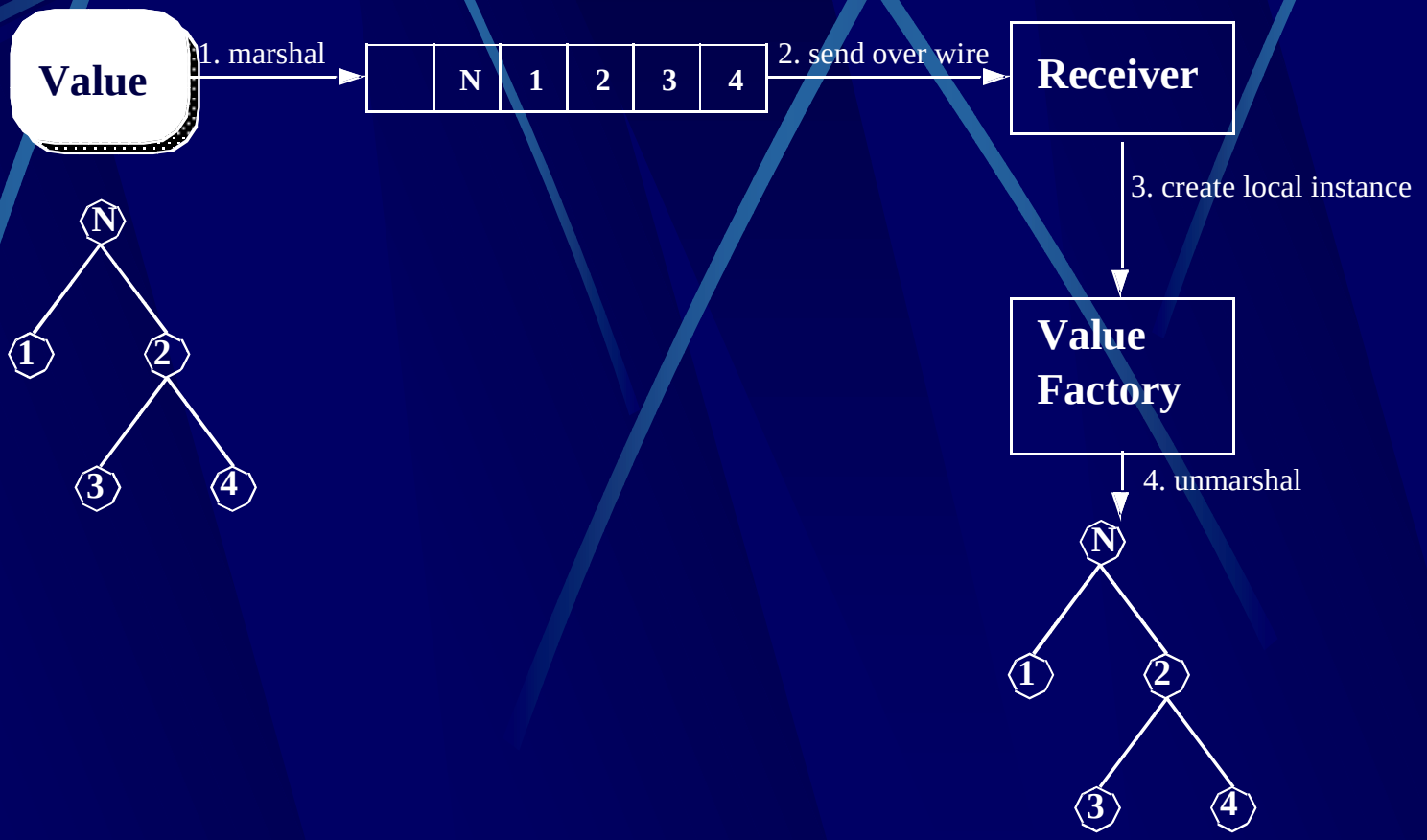
Object by Value

- CORBA introduces a new value type which inherits from `CORBA::ValueBase`:

```
value BinaryTree {
        long value;
        BinaryTree left;
        BinaryTree right;
        // initializer
        init(in long w);
        // local operations
};
```

- A value has no IOR and cannot be accessed remotely. However, value types might also inherit from other CORBA interfaces.

Object by Value (cont'd)



Object by Value (cont'd)

- Values can define any recursive (cyclic) structures that might be null.
- Value types can be single inherited from other value types.
- They are local to the receiver and are marshalled when transmitted across the wire.
- Benefits of ObV:
  - Values allow to exchange complex state.
  - Consistent semantics across different programming languages.
  - Natural support for C++ and Java.
  - Minimal impact on GIOP/IDL.
  - Enables RMI over IIOP.

CORBA ORB Interfaces

- The *ORB interface* contains functionality that might be required by clients or servers.
- The *Dynamic Invocation Interface* provides a means for dynamically invoking CORBA objects that were not known at design-time.
- The *Dynamic Skeleton Interface* helps to implement generic CORBA servants.
- The Basic *Object Adapter* is the API used by the servers to register their object implementations. In addition, it is the immediate layer between the ORB Core and the IDL skeleton.

ORB Interface

The ORB interface mainly provides helper functions to clients and servers.

```
interface ORB {
        string object_to_string(in Object obj);
        Object string_to_object(in string str);
        …
}
```

Dynamic Invocation

- Dynamic Invocation allows to invoke servers without linking stub code.
- Each CORBA interface is derived from the *interface Object* and therefore needs to implement the method:

```
ORBStatus create_request(
                in Context ctx, // context object
                in Identifier operation, // operation
                in NVList arg_list, // arguments
                inout NamedValue result, // result
                out Request request, // new request
                in Flags req_flags // flags
)
```

Dynamic Invocation (cont'd)

- Example:

```
CORBA::Float f;
CORBA::Object_ptr obj = ...
CORBA::Request_ptr req = obj->_request("method");
req << CORBA::outMode << f;
req->send_deferred();
// some time later:
req->get_response();
CORBA::Float res;
req >> res;
```

Dynamic Skeleton Interface

- The *DSI* corresponds to the *Dynamic Invocation Interface*. Used when a server wants to dispatch requests itself: CORBA Bridges, Debuggers, Interpreter Environments.
- For this purpose, a generic interface is available: *Dynamic Invocation Routine*:

```
module CORBA {
    interface ServerRequest {
        Identifier op_name(); // operation name
        OperationDef op_def(); // operation definition
        Context ctx();
        void params(inout NVList params);
        NamedValue result;
    };
    ...
```

CORBA Messaging

- **CORBA Messaging extends CORBA with asynchronous method invocations:**
- **Asynchronous Method Invocation (AMI) allows to decouple client from server operation (non-blocking communication). There are two models: *Polling model*, *Callback Model*.**
- **“Store and forward” semantics supported by TII (Time-Independent Invocations). Invocations might outlive client process. For this purpose, IRP (Interoperable Routing Protocol) based upon GIOP is introduced. Integration into existing MOM products possible.**
- **Quality of Service on thread, object reference or ORB level. For instance, timeouts, priority and ordering, rebinding, “store-and-forward”, ...**

CORBA Messaging (cont'd)

- Callback Model Sample:

```
exception IDoNotLikeToSpeak;
interface Talk {
   string talk(in string msg) raises(IDoNotLikeToSpeak);
};
// the IDL compiler will implicitely treat this as:
value AMI_TalkExceptionHolder : Messaging::ExceptionHolder {
   void raise_talk() raises (IDoNotLikeToSpeak);
};
interface AMI_TalkHandler : Messaging::ReplyHandler {
   void talk(in string ami_return_val);
   void talk_excep(in AMI_TalkExceptionHolder eh);
};
exception IDoNotLikeToSpeak;
interface Talk {
   string talk(in string msg) raises(IDoNotLikeToSpeak);
   void sendc_talk(in AMI_TalkHandler h, in string msg);
};
```

SIEMENS CORBA

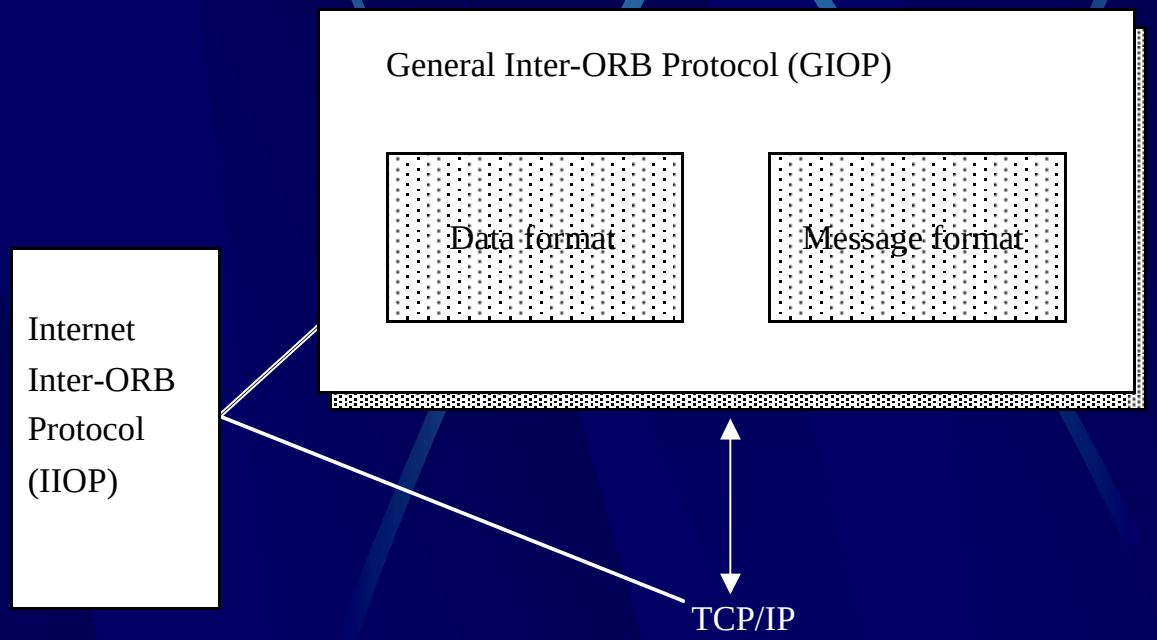
OOP 2001 MÜNCHEN

Initialization

- Question: How do I connect to my CORBA system?
- The following steps are necessary:
  - `CORBA::ORB_ptr orb = ORB_init(..., ORBID)` will return a pseudo object for accessing the broker.
  - `orb->resolve_initial_references(params)` will return fundamental objects such as the COSS Naming Service depending on the parameter passed.

ORB Interoperability

- If two ORBs cooperate, they need a common language.



GIOP-Protocol

GIOP (General Inter-ORB Protocol) implementations consist of:

- **The *Common Data Representation:* transfer syntax from IDL to low-level representation (byte ordering, aligned primitive types, mapping for IDL types).**
- *GIOP Message Formats*: Format of messages exchanged between ORBs such as Request, Reply, Fragment, CancelRequest, LocateRequest, LocateReply, CloseConnection, MessageError, LocationForward.
- *GIOP Message Transport*: Designed to work on various transport protocols that are connection-oriented, reliable, can be considered as byte stream, notify about disorderly connection lost, model for initiating connections can be mapped on the TCP/IP model.

IIOP

- *The Internet IOP Message Transport* describes how agents open TCP/ IP connections and use them to transfer GIOP messages.
- **IIOP is not a separate specification but a specialization and mapping of the GIOP for TCP/IP.**

```
module IIOP { // Definition of IOR
        struct Version { char major; char minor; };
        struct ProfileBody {
        Version iiop_version;
        string host; unsigned short port;
        sequence<octet> object_key;
};
```

SIEMENS CORBA

OOP 2001 MÜNCHEN

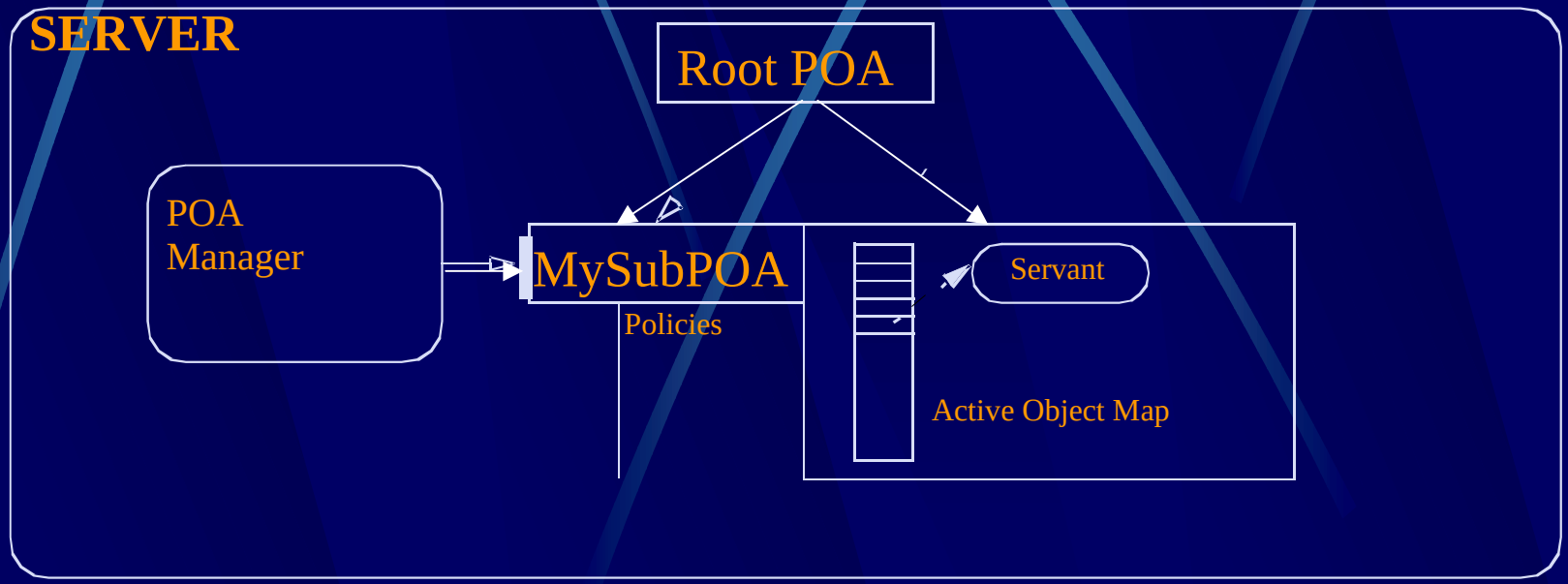
IOR

- Bridges need information on object references: Is it null? What type is it? What protocols are supported? What ORB Services are available?
- IORs (Interoperable Object References) are introduced to integrate this information. They are not visible to programmers.
- IORs contain a type id and a tagged profile per protocol supported (needed by the protocol to identify an object).
- IORs are created from object references before the request crosses a domain boundary.
- IORs can be stringified / destringified.

Portable Object Adapter

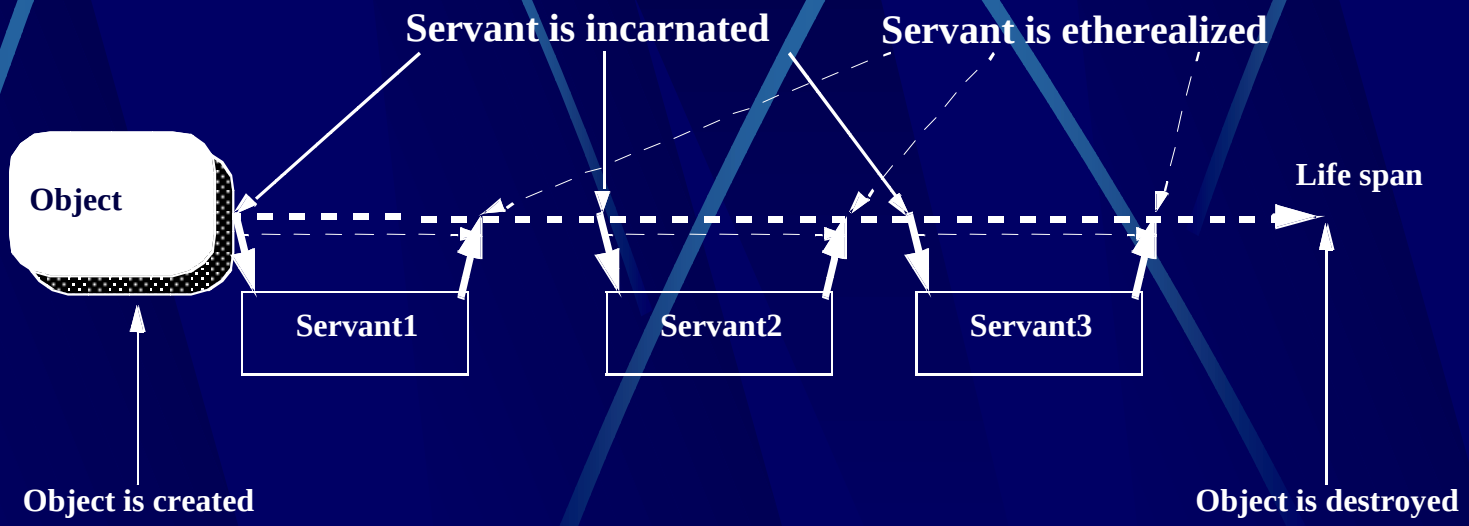
- The Portable Object Adapter overcomes the limitations of the BOA.
- A POA instance is the place where CORBA objects and servants live.
- *Servants* are running implementations of “virtual“ CORBA objects.
- Policies specify how servants map to object references.
- POAs are the places where CORBA objects live. All CORBA objects within a POA share the same policies. POAs may be nested in a tree structure.
- Object IDs identify servants within their POA.
- POAs manage, activate and deactivate servants.

POA Hierarchy



Servants and Objects

- CORBA objects are implemented by servants.



- *Transient* CORBA objects do not survive their creator process.
- *Persistent* CORBA object persist across multiple processes.

POA Responsibilities

- **POAs are responsible for ...**
  - ***creating object references.***
  - ***uniquely identifying objects by Object IDs where either the POA or the implementation can supply the Object IDs.***
  - ***managing the servants which are registered by an application. It stores all servants in an Active Object Map. All requests where no servant exists, can be routed to a user-defined default servant or to a user-defined servant-manager.***
- ***When a request arrives the target ORB dispatches it to the POA hosting the target object.***

POA Policies

- Each POA defines its own set of policies- Root POA has standard set.
- POAs are derived from existing POAs, but do not inherit any policies.
  - Thread (ORB_CTRL_MODEL, SINGLE_THREAD_MODEL)
  - Lifespan (TRANSIENT, PERSISTENT)
  - Object ID Uniqueness (UNIQUE_ID, MULTIPLE_ID)
  - ID Assignment (USER_ID, SYSTEM_ID)
  - Servant Retention (RETAIN, NON_RETAIN)
  - Requests (USE_ACTIVE_OBJECT_MAP_ONLY, USE_DEFAULT_SERVANT, USE_SERVANT_MANAGER)
  - Implicit Activation (IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION)

Servant Managers

- The user-supplied Servant Manager is invoked by the POA if the POA cannot find a servant implementing a requested object.
- This happens, for instance, if the server process wants to create references at start-up, but incarnate servants only on demand.
- Two types of Servant Managers:
  - *Servant Activator* is used when RETAIN policy is used. Servant is inserted in Active Object Map.
  - *Servant Locator* is used when NON_RETAIN is used instead. Servant is used for invocation only and is not retained.

SIEMENS CORBA

OOP 2001 MÜNCHEN

Default servants

- Default servants are used to incarnate objects when USE_DEFAULT_SERVANT policy or NON_RETAIN policy are applied.
- They are used to incarnate multiple objects with the same interface, e.g., objects providing the Dynamic Skeleton Interface.
- They must not hold any object-specific state.

POA Manager

- With each POA a POA Manager is associated.
- The POA Manager allows to:
  - *activate* a POA (start its work)
  - *deactivate* a POA (stop its work)
  - *hold requests* (incoming requests are queued but not executed)
  - *discard requests* (incoming and queued requests are discarded)

POA Sample Code

```
CORBA::ORB_var orb = CORBA_ORB_init(argc, argv, “myORB”);
CORBA::Object_var rootPoaObj = orb->resolve_initial_references(“RootPOA”);
PortableServer::POA_var poa = PortableServer::POA::_narrow(rootPoaObj);
PortableServer::POAManager_var mgr = poa->the_POAManager();
CORBA::PolicyList polList;
polList.length(1);
polList[0]
 = poa->create_request_processing_policy(PortableServer::USE_SERVANT_MANAGER);
PortableServer::POA_var myPoa = poa->create_POA(“MyPoa”, mgr, polList);
MyObjectManager myObjectMgr;
PortableServer::ServantManager_var mySerMgr = myObjectMgr._this();
myPoa->set_servant_manager(mySerMgr)
MyObject_impl * servant = new MyObject_impl(myPoa);
PortableServer::ObjectId_var oid = myPoa->activate_object(servant);
mgr->activate();
```

SIEMENS CORBA

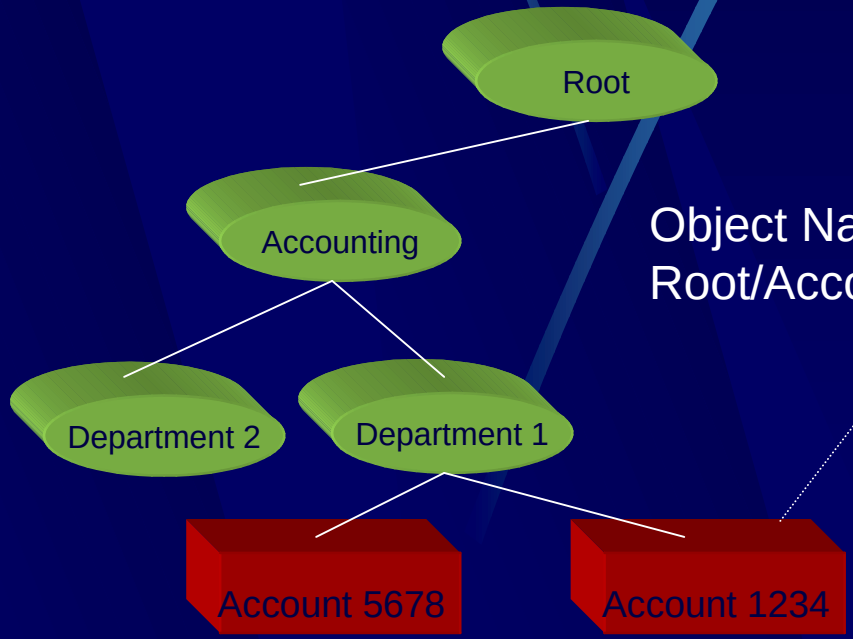
OOP 2001 MÜNCHEN

Is that all we need?

- So far we have introduced a messenger component called the Broker that locates servers and feeds them with client requests.
- Real-world applications need more than just a Remote Method Invocation paradigm.
- We need to use advanced services such as Events, Naming, Database Access, Transaction Processing, Lifecycle Support, ...
- Fortunately, the OMG already has standardized a rich set of additional services with even more to come.
- CORBAservices are fundamental services provided by IDL interfaces and CORBA objects. These services aim at the physical modeling and storing of objects.

Naming Service

- Question: Where do you get object references?
- Answer: Use the Naming Service
- The Naming Service is similar to a file system directory:



Object Name:
Root/Accounting/Department 2/Account_1234

Naming Service (cont'd)

Fundamental concepts:

- **A *name binding* (name, object) associates a name with an object.**
- **A *naming context* denotes a set of name bindings with each name being *unique*. Name contexts themselves are CORBA objects. Hence, they can also be bound to a name.**
- **Name bindings are always relative to a naming context.**
- **A name is a sequence of *name components* each of them consisting of an identifier and a kind:** Root/Accounting/Department 2/Account_1234

Naming Service (cont'd)

Some methods from the CosNaming module:

```
module CosNaming { // ...
      interface NamingContext {
       void bind(in Name n, in Object obj) raises ...;
       void rebind(in Name n, in Object obj) raises ...;
       void bind_context(in Name n, in NamingContext nc) raises ...;
       void rebind_context(in Name n, in NamingContext nc) raises ...;
       Object resolve(in Name n) raises ...;
       void unbind(in Name n) raises ...;
       NamingContext new_context();
       NamingContext bind_new_context(in Name n) raises ...;
       void destroy() raises ...;
      };
};
```

Naming Service (cont‘d)

Example Code:

```
// Initialize the ORB pseudo object:
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
// get initial reference to the name service
CORBA::Object_var obj = orb->resolve_initial_references(“NameService”);
// use initial context:
CosNaming::NamingContext_var root = CosNaming::NamingContext::_narrow(obj);
CosNaming::Name n;
n.length(1);
n[0].id = CORBA::string_dup(“MyObject1”);
// and try to resolve name
CORBA::Object_var objInDir = root->resolve(n);
// narrow to object type
MyType_var myObj = MyType::_narrow(objInDir);
```

SIEMENS CORBA

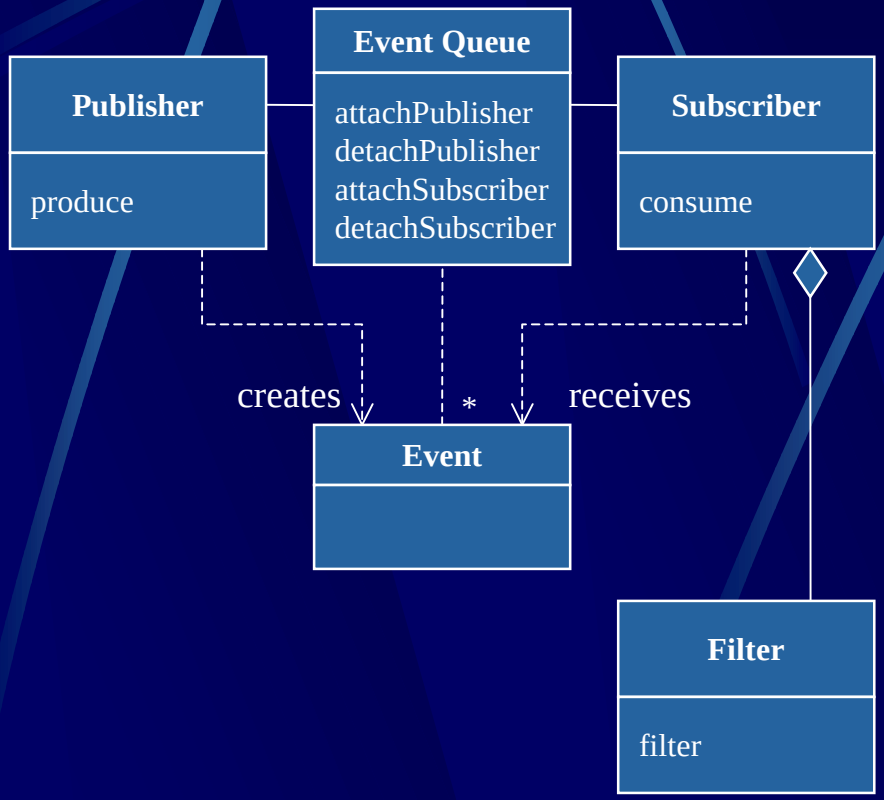
OOP 2001 MÜNCHEN

Event Service

- Usually, a specific client calls a specific remote server and blocks until the result returns.
- Sometimes, this strategy is not sufficient.

  Consider a server that reports share values.
  - A polling strategy leads to performance bottlenecks.
  - The share values could be spread across different servers.
  - More than one client may be interested in the information.
- How can we decouple clients and servers?

Event Service (cont'd)



Decouple suppliers (publishers) and consumers (subscribers) of events:

An *Event Queue* is storing events.

Publishers create events and store them in an event queue with which they have previously registered.

Consumers register with event queues from which they retrieve events.

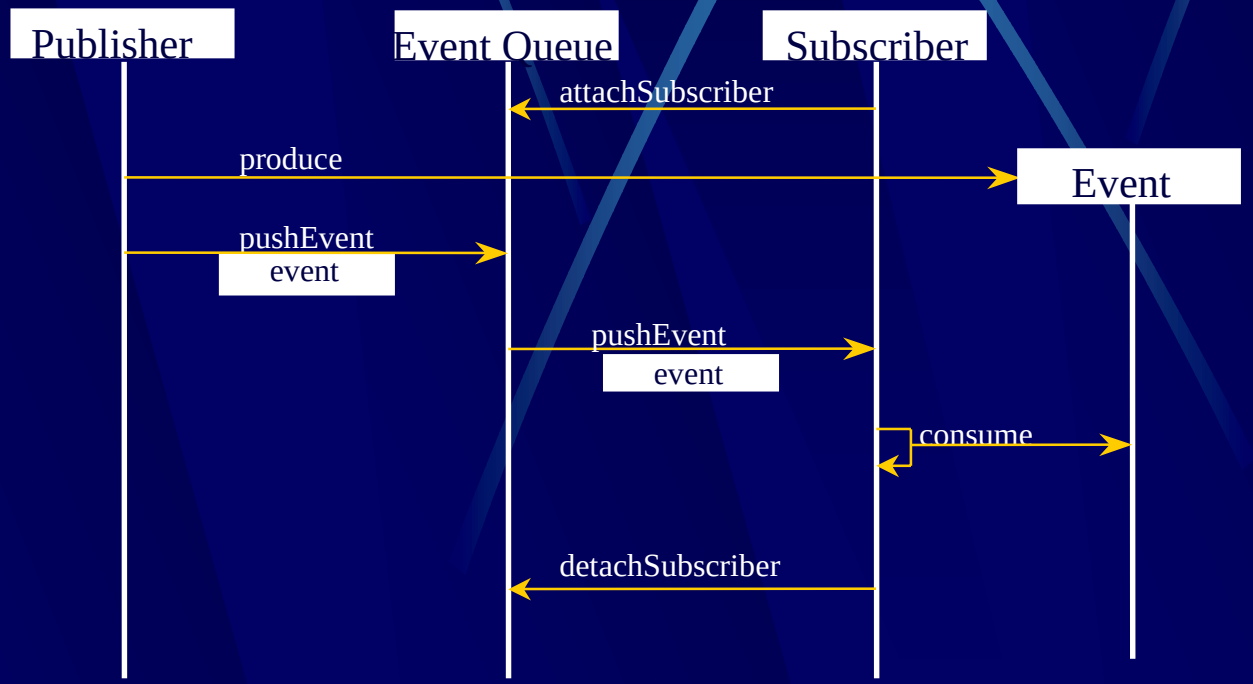
Events are objects used to transmit state change information from publishers to consumers.

For event transmission *push-models* and *pull-models* are possible.

Filters could be used to filter events on behalf of subscribers.

Event Service (cont'd)

- Dynamics (simplified)



Event Service (cont‘d)

- Some characteristics:
- Event consumers and event suppliers are decoupled from each other.
- A many-to-many relationship between consumers and suppliers is supported.
- Push-style as well as Pull-style communication is available.
- Typed and untyped events are possible.
- The *Notification Service* extends the Event Service. It enables developers to add filters to event channels.

CORBA 3

- In the year 2000 the OMG has published CORBA 3 which offers solutions in 3 areas:
  - Internet
  - Quality of Service
  - Components

SIEMENS CORBA

OOP 2001 MÜNCHEN

Internet

- The Interoperable Name Service (INS) specifies URL-based naming schemes such as
  - `iioploc://1.1@mymachine.siemens.de:9999/bin/tradingservice`
  - `iiopname://mymachine.siemens.de/root/accounting/departmentA/a1`
- Java Reverse Mapping allows to omit usage of CORBA IDL. Developers just follow the RMI conventions.
- Bidirectional GIOP allows clients and servers to share the same connection in contrast to traditional GIOP.
- CORBA Firewall introduces GIOP Proxies.

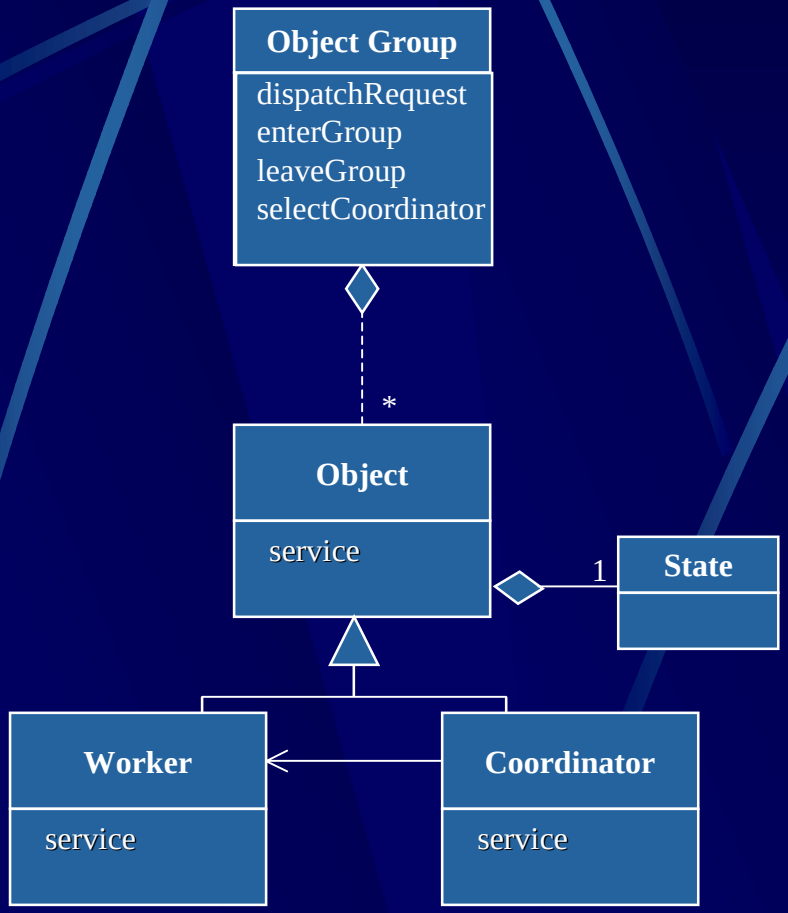
Quality of Service

- CORBA Messaging defines asynchrnous calls (AMI) as well as time-independent calls (TII)
- Minimum CORBA is CORBA without all dynamic features such as
  - DII
  - DSI
  - DynAny
  - Interface Repository
  - Some of the POA policies and options

Quality of Service (cont'd)

- Some facts about Realtime CORBA:
  - Threads are dispatched by a special dispatcher that supports static scheduling.
  - Configuration is done using `RT_CORBA::RTORB`.
  - Thread Pools can be created and thread priorities changed.
  - RT CORBA supports a specific priority scheme that is mapped to the native scheme.
  - Priority inversions are prevented by different strategies such as sending priorities using IIOP, synchronizing resource access by mutexes.
  - Clients can initiiate different connections to servers such as channels with different priorities, dedicated connections, and specify time-outs for invocations.

Quality of Service (cont'd)



Fault toleranct CORBA:

An *Object Group* represents a group of objects that all provide the same service.

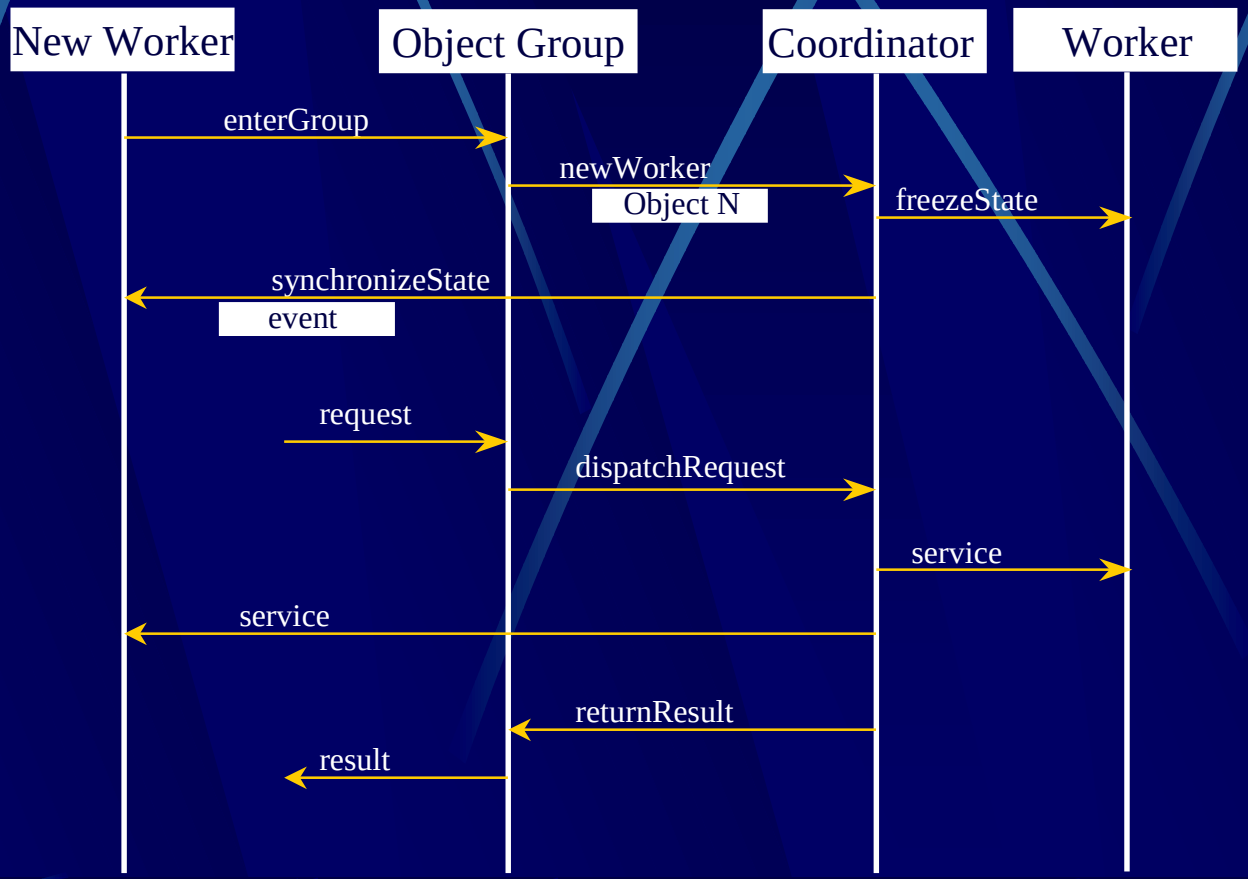
Object provides a specific service to clients.

The *coordinator* is selected by the Object Group to dispatch requests.

Workers receive requests from their coordinator and return results.

Each member of the object group maintains its own *state*.

Quality of Service (cont'd)



Quality of Service (cont'd)

- Some facts about Fault-tolerant CORBA:
  - FT CORBA supports reliability by replication.
  - Replicated objects are part of an object group which behaves to the clients as if it were a single object. Thus, client is oblivious to FT mechanisms.
  - Object groups are created by replication managers. There can be more than one replication manager per host.
  - Local and global agents are responsible for detecting error conditions.
  - Errors are reported to the reoplication manager using the fault notifier.

Quality of Service (cont'd)

- More facts about Fault-tolerant CORBA:
  - There are two options for handling error conditions:
    - In active configurations all members of an object group receive all requests in the same order using a reliable multicast protocol. If only one object succeeds, a result can be returned to the client.
    - In passive configurations a primary object handles the request. If the primary object fails, the system uses a backup object instead. The state synchronization is committed either:
      - after each request (warm passive replication) *or*
      - using a recovery file (cold passive replication).
    - All activities are logged by the system.

SIEMENS
CORBA

OOP 2001
MÜNCHEN

CORBA Components

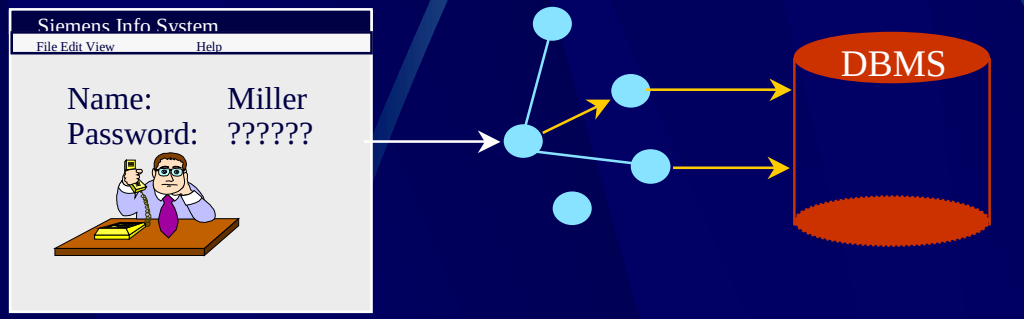
- What you will learn in this part:
  - Overview of CORBA Components
- What you will not learn:
  - How to design and implement CORBA components
  - Concrete Language Mappings
- Note: CORBA Components Spec. is currently under construction!

Motivation

- Why Components?

Component Software

- Multi-tier architectures lead to separation of concerns (presentation, application, data).
- But presentation / application tiers are complex.
- Thus, an additional separation of these tiers is necessary!



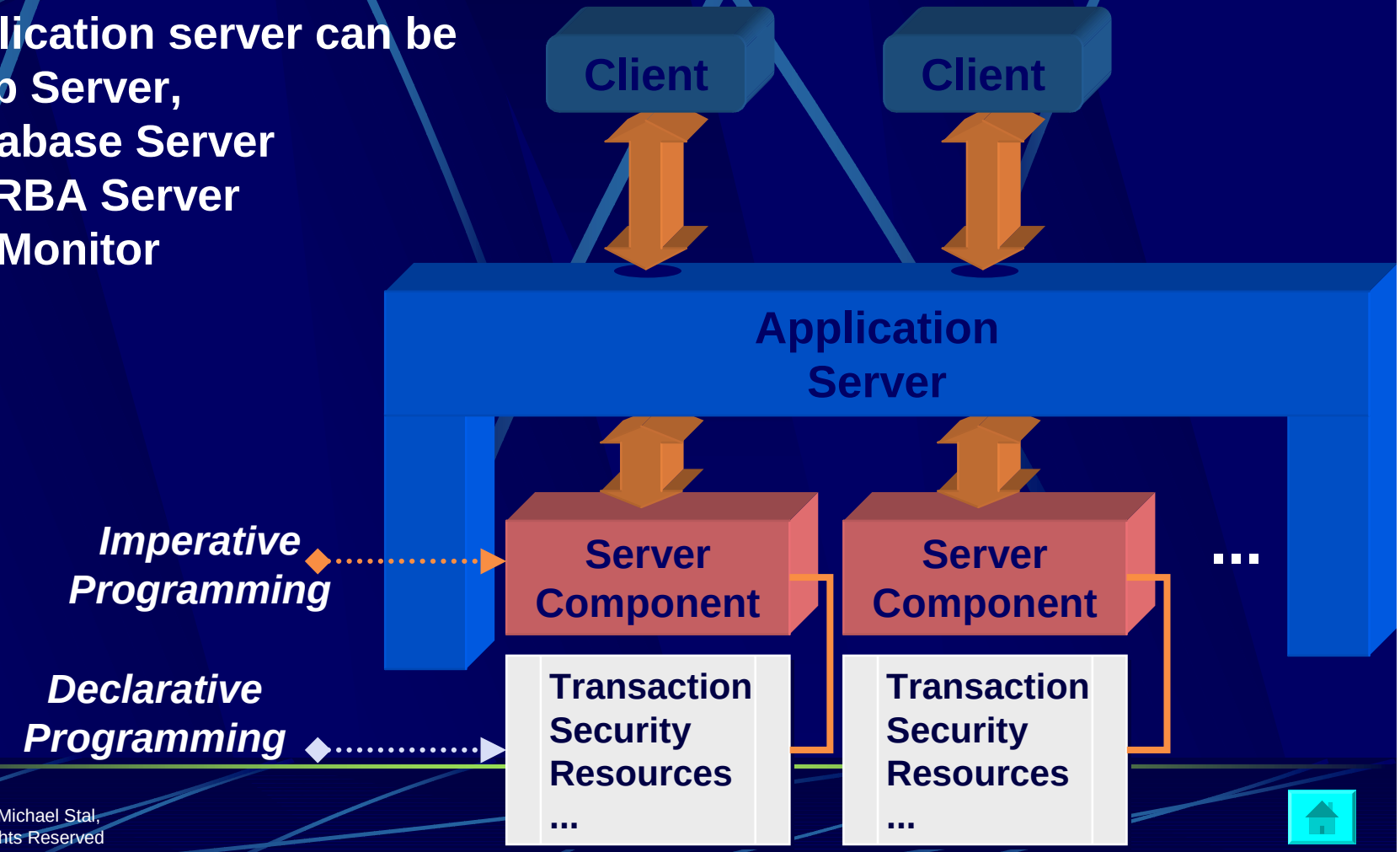
Flavors of Components

- Presentation tier components:
  - they typically represent sophisticated GUI elements.
  - they share the same address space with their clients.
  - their clients are containers that provide all the resources.
  - they send events to their containers.
- Middle tier components:
  - they typically provide server-side functionality.
  - they run in their own address space.
  - they are integrated into a container that hides all system details.

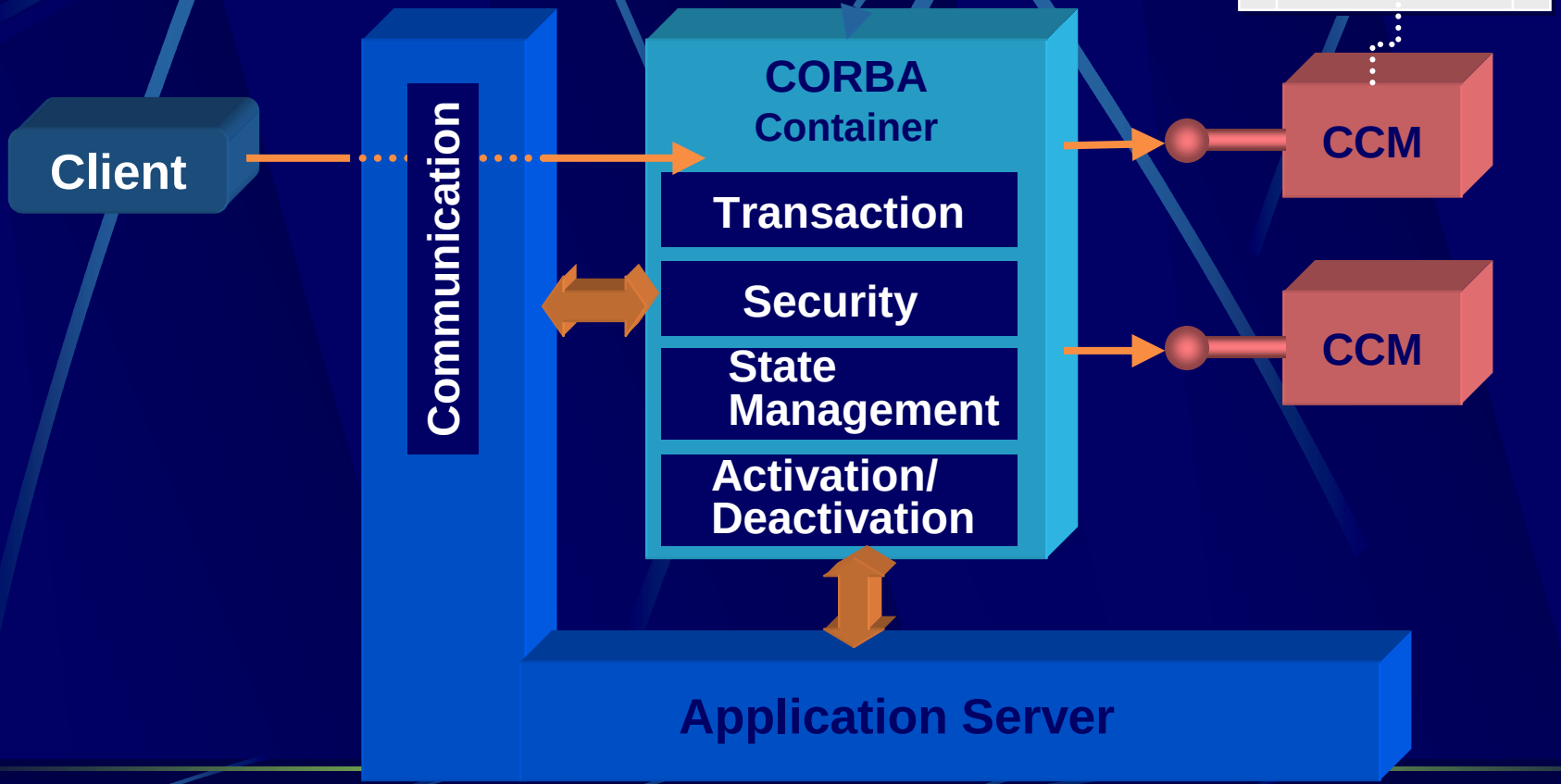
Application Servers

An application server can be
- **a Web Server,**
- **a Database Server**
- **a CORBA Server**
- **a TP-Monitor**

...



CORBA based Application Server



SIEMENS
CORBA

OOP 2001 MÜNCHEN

CORBA Components – EJB made Ubiquitious

Client Usage Example

```
// Get home finder:
org.omg.CORBA.Object objref =
   orb.resolve_initial_references(„ComponentHomeFinder“);
// „cast“ it to correct type:
ComponentHomeFinder hf = ComponentHomeFinderHelper.narrow(objref);
// find home using type:
org.omg.CORBA.Object of = hf.find_home_by_type(BankHomeHelper.id());
// „cast“ it to correct type:
BankHome bh = BankHomeHelper.narrow(of);
// create instance and narrow it:
org.omg.Components.ComponentBase bankInstance = bh.create();
Bank myBank = BankHelper.narrow(bankInstance);
// invoke operations:
long how_much_money = myBank.amount();
```

Equivalent IDL

- All component definitions are specified using Component IDL.
- Client mappings are described using equivalent IDL.
- Thus, all features can be described with CORBA 2.3 compliant IDL grammar.

Equivalent IDL (cont'd)

- For example,

```
component MyComp { ... };
```

- is equivalent to

```
interface MyComp
: Components::ComponentBase { ... };
```

BTW, this is the distinguished interface of the CORBA component!

Supported Interfaces

- Similar to Java, a component (interface) may support other interfaces:

```
interface  WhoAmI {
        String getName();
};

component UnixUser supports WhoAmI {
}
```

Inheritance

- Components may singly inherit from other components:

```
component BaseComponent  supports A {
        // implement A methods
        // implement further methods
};

component DerivedComponent : BaseComponent supports B {
        // implement BaseComponent methods incl. A methods
        // implement B methods
        // implement further methods
};
```

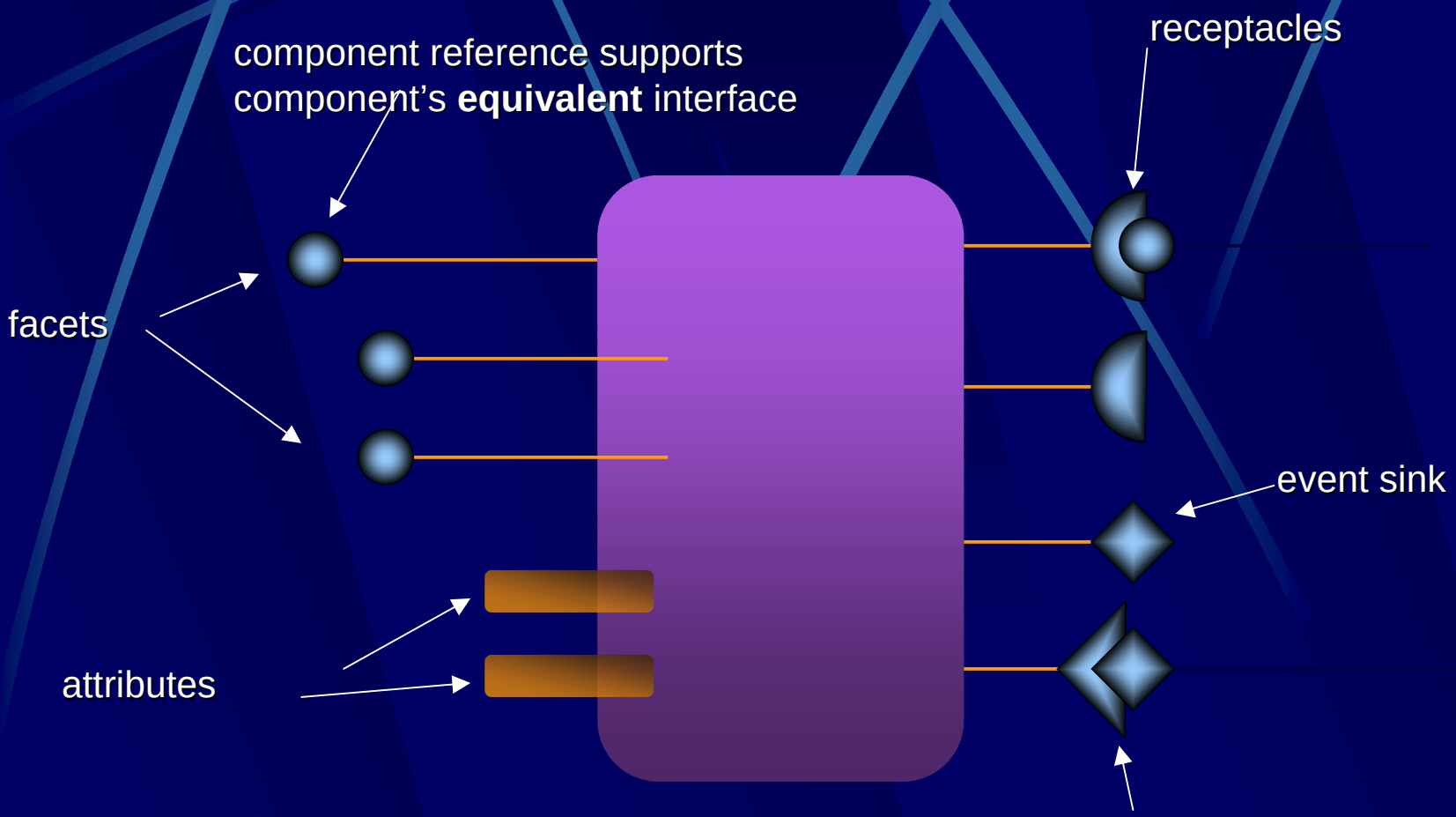
SIEMENS CORBA

OOP 2001 MÜNCHEN

Ports

- Components specify interfaces (ports) to interact with their environment:
  - Facets are provided interfaces for clients
  - Receptacles denote connection points
  - Event Sources
  - Event Sinks
  - Attributes for configuration purposes

Ports (cont‘d)



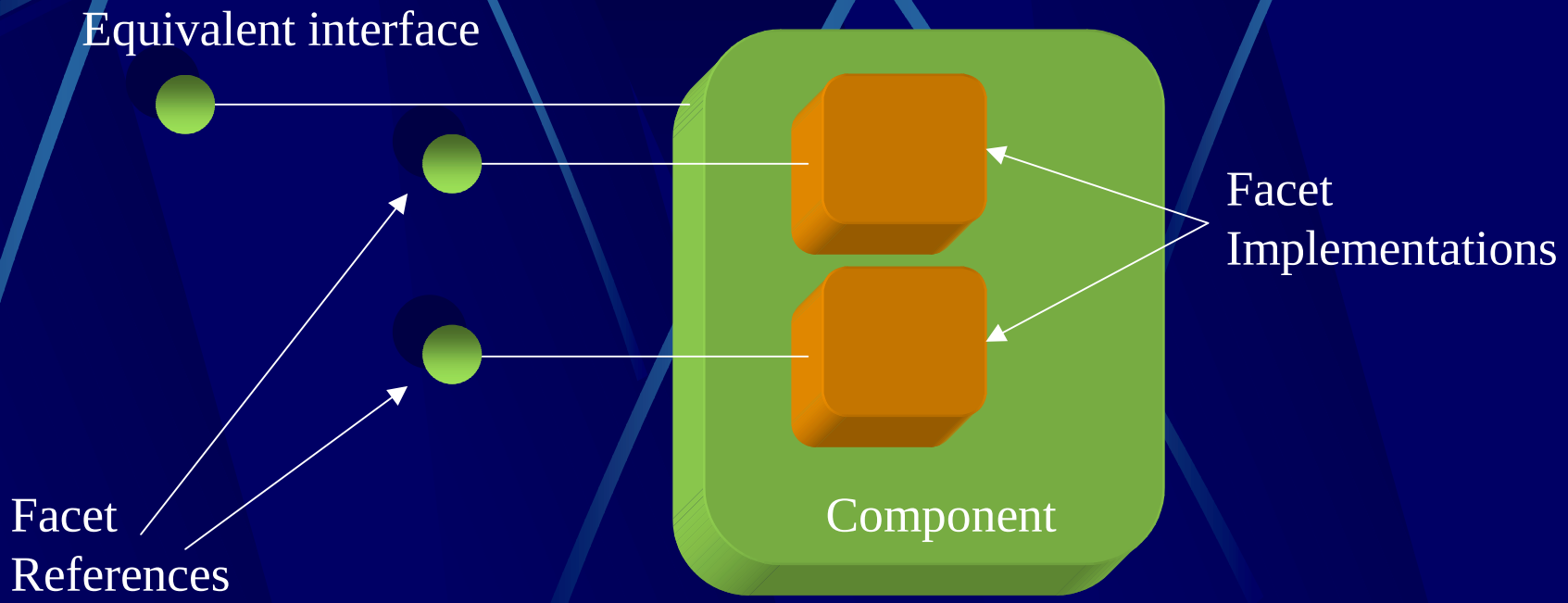
SIEMENS CORBA

OOP 2001 MÜNCHEN

Facets

- Each component has a single distinguished equivalent interface.
- It may additionally provide multiple object references (facets).
- Equivalent Interface used by clients for navigation and connection.

Facets (cont‘d)



Facets (cont'd)

- Facet implementations encapsulated by components; opaque to clients.
- Clients navigate from facet to equivalent interface (`get_component()`) and ...
- obtain facets from equivalent interface (`provide...` methods).

Facets (cont'd)

- Component IDL:

```
interface Invoice { ... };
interface Customer { ... };

component ShoppingTour {

        provides Invoice the_invoice;
        provides Customer the_customer;
};
```

- Equivalent IDL:

```
interface Invoice { ... };
interface Customer { ... };

interface  ShoppingTour :
        Components::ComponentBase {
        Invoice provide_ the_invoice();
        Customer provide_the_customer();
};
```

Receptacles

- Receptacles denote component's ability to use other object (references).
- When component accepts object reference, this is called a connection.
- Typically, connections are set during assembly.

Receptacles (cont'd)

- Component IDL:

```
interface Printer  { ... };

component ShoppingTour {

        uses Printer the_printer;
};
```

- Equivalent IDL:

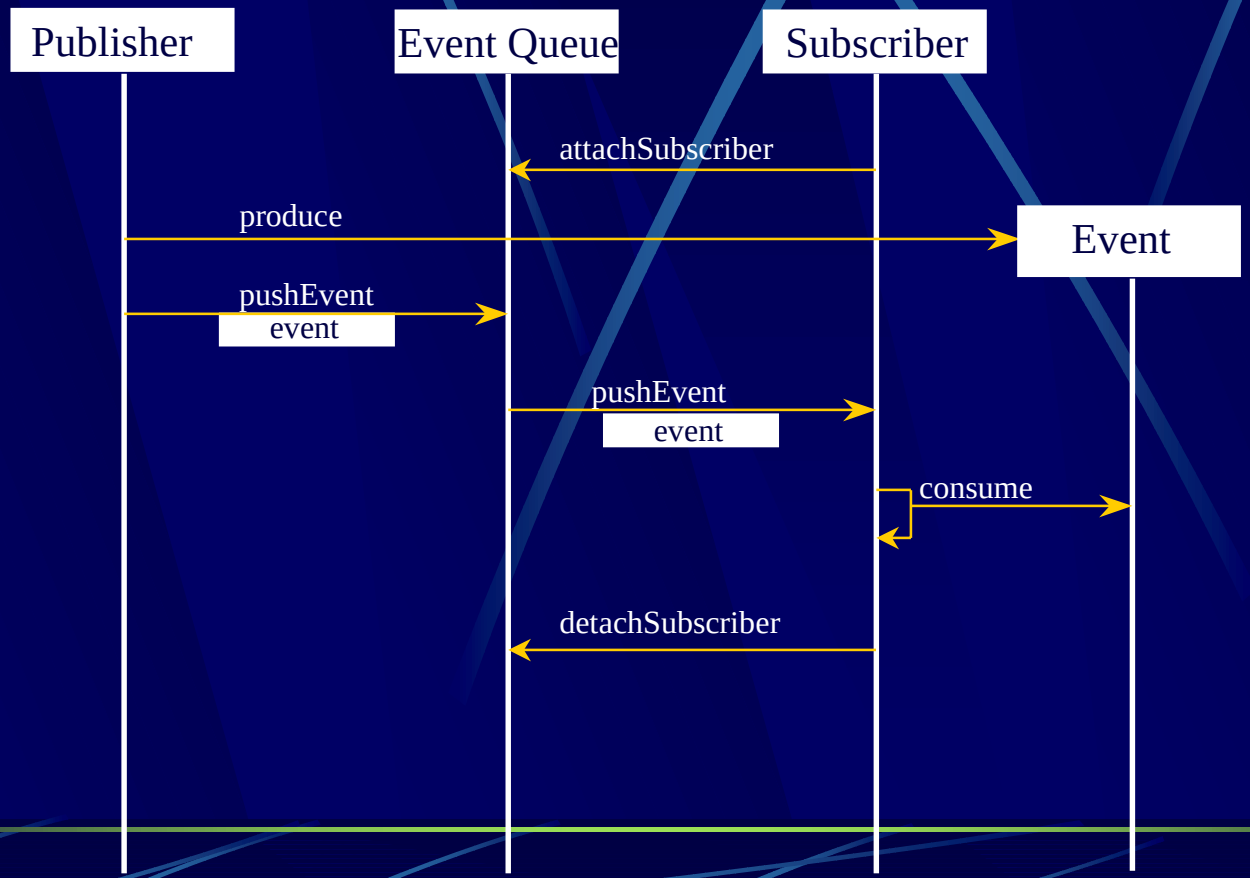
```
interface Printer { ... };

interface  ShoppingTour :
        Components::ComponentBase {
        void
          connect_the_printer(in Printer co);
        Printer
          disconnect_the_printer();
        Printer
          get_connection_the_printer();
};
```

- Keyword „multiple“ to allow multiple interfaces of the same type

Events

- Components can send and receive event notifications.
- An *emitter* sends *events* to exactly one *subscriber*. A *publisher* is not restricted to one subscriber.
- CORBA Push Model used; value types are sent as an **any**.
- Container responsible for QoS and channel management.

Publisher/Subscriber Pattern



Publishers and Emitters

- Publisher
  - pushes events to multiple consumers vin event channel.
  - Subscribers delegated to event channel.
  - Component is only source.
- Emitter
  - can be connected to at most one subscriber
  - direct registration with event source
  - direct push to subscriber

Publisher Example

- Component IDL:

```
valuetype Limit :
        Components::EventBase
        { ... };

component ShoppingTour {
        publishes
        Limit lim;
};
```

- Equivalent IDL:

```
valuetype Limit:
        Components::EventBase { ... };
module ShoppingTourEventConsumers {
        interface LimitConsumer :
                ComponentsEventConsumerBase {
                void push(in Limit evt);
        };
};
interface  ShoppingTour :
        Components::ComponentBase {
        Components::Cookie
          subscribe_lim (in
            ShoppingTourEventConsumers::LimitConsumer
                c);
         ShoppingTourEventConsumers::LimitConsumer
          unsubscribe_lim (in Components::Cookie ck);
};
```

Emitter Example

- Component IDL:

```
valuetype SingleOffer :
        Components::EventBase
        { ... };

component ShoppingTour {
        emits
        SingleOffer prd;
};
```

- Equivalent IDL:

```
valuetype SingleOffer : Components::EventBase { ... };
module ShoppingTourEventConsumers {
        interface SingleOfferConsumer :
                        ComponentsEventConsumerBase {
                        void push(in SingleOffer evt);
        };
};
interface  ShoppingTour :
        Components::ComponentBase {
        void connect_prd (in
            ShoppingTourEventConsumers::
            SingleOfferConsumer  c);
         ShoppingTourEventConsumers::
         SingleOfferConsumer
             disconnect_prd ();
};
```

Subscribers

- A subscriber can be connected to any number of event sources.
- Externally established connections.

Subscriber Example

- Component IDL:

```
valuetype SingleOffer :
        Components::EventBase
        { ... };

component Customer {
        consumer
        SingleOffer prd;
};
```

- Equivalent IDL:

```
valuetype SingleOffer : Components::EventBase { ... };
module CustomerEventConsumers {
        interface SingleOfferConsumer :
                ComponentsEventConsumerBase {
                void push(in SingleOffer evt);
        };
};
interface  Customer :
        Components::ComponentBase {

        CustomerEventConsumers::
         SingleOfferConsumer
            get_consumer_prd ();
};
```

Attributes

- Used primarily for configuration purposes:
  - Customization of components
  - Support by visual tools
  - Homes can apply collections of settings to all components.

Component Identity

- Components identified by component reference.
- Operations to check whether two references belong to same component.
- „Sameness“ is up to implementor.
- Components may be associated with primary keys.

SIEMENS CORBA

OOP 2001 MÜNCHEN

Component Homes

- Component home meta-type.
- Component home responsible for managing instances of *one* type.
- Operations to manage component life-cycles, and (optionally) primary keys.
- Homes and Components are defined in isolation.

Home Example

- Component IDL:

```
home ShoppingTourHome
        manages ShoppingTour
{
        // explicit operations
};
```

- Equivalent IDL:

```
interface ShoppingTourHomeImplicit {
        ShoppingTour create();
};
interface ShoppingTourHomeExplicit: {
        // explicit operations
};
interface ShoppingTourHome :
        ShoppingTourHomeImplicit,
        ShoppingTourHomeExplicit,
        HomeBase // Life-Cycle Operations
{
        ...
};
```

Primary Keys

- A home can provide a primary key to uniquely identify components:
  - Used to find/remove components
  - assigned on creation, or through database
  - valuetype derived from `Components::PrimaryKeyBase`
  - Primary key usually not part of component
  - Different homes may use different primary keys for same component type

Primary Key Example

- Component IDL:

```
valuetype Key :
Components::PrimaryKeyBase {
        public long value;
};

home ShoppingTourHome
        manages ShoppingTour
        primaryKey Key
{
                // explicit operations
};
```

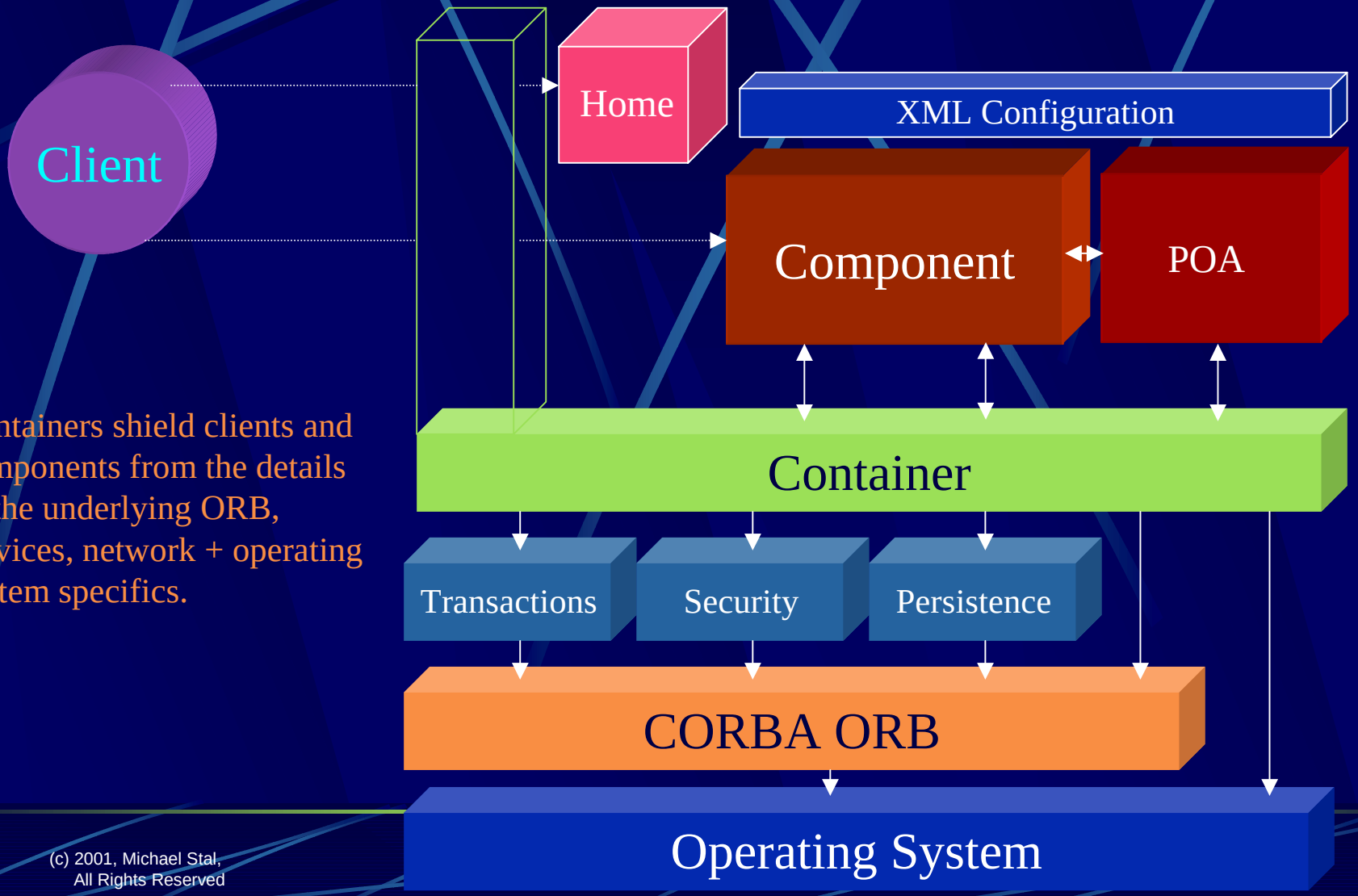
- Equivalent IDL:

```
interface ShoppingTourHomeImplicit {
        ShoppingTour create(in Key k);
        ShoppingTour find_by_primary_key(in Key k);
        void remove(in Key k);
        Key get_primary_key(in ShoppingTour st);
};
interface ShoppingTourHomeExplicit {
        // explicit operations
};
interface ShoppingTourHome :
        ShoppingTourHomeImplicit,
        ShoppingTourHomeExplicit,
        ComponentHome // Life-Cycle Operations
{
        ...
};
```

Home Finders

- But how does a client find a home?
  - Homes register with home finders.
  - Clients use home finders to obtain homes.
  - Home finder might select one of several homes using client request and configuration data.
  - Home finder itself is obtained using `ORB::resolve_initial_references`

Containers are a developers's best friend

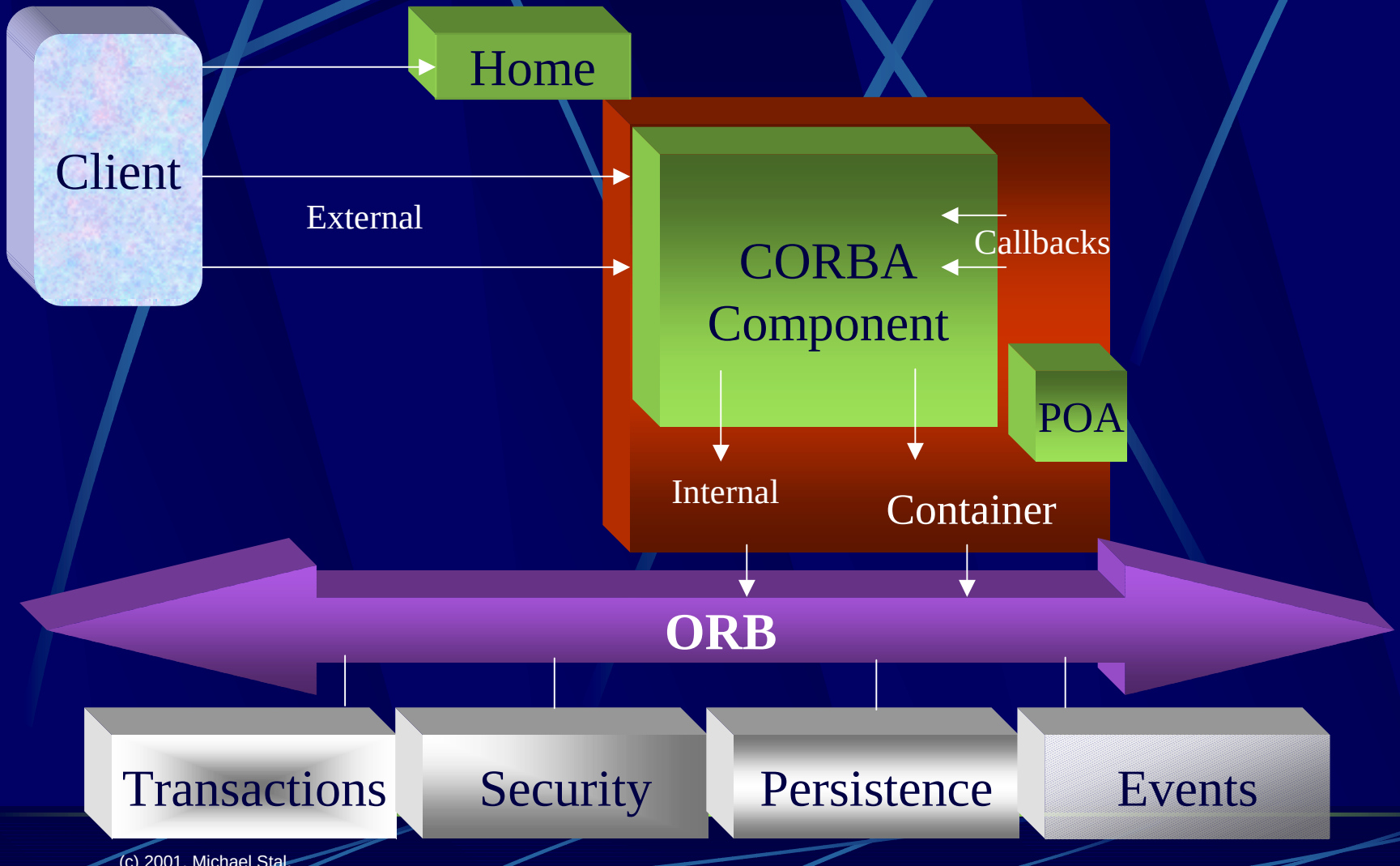


Containers shield clients and components from the details of the underlying ORB, services, network + operating system specifics.

Container Programming Model

- Container is a runtime environment for a CORBA component implementation.
- Environment may be a deployment platform (app server) or IDE.
  - IDE: Highly customizable, no concurrent users
  - App. Server: Many concurrent users, but not highly customizable.

Overall Architecture



SIEMENS CORBA

OOP 2001 MÜNCHEN

Elements

- External Types
- Container Type
- Container Implementation Type
- Component Category

SIEMENS CORBA

OOP 2001 MÜNCHEN

External Types

- Contract between component and client
- Home interfaces to obtain interface references and application interfaces
- Two design patterns supported:
  - Home with primary key: *finder* + *factory*
  - Home without primary key: *factory*

~ analogous to EJBHome, EJBObject

SIEMENS CORBA

OOP 2001 MÜNCHEN

Container Type

- API framework between component and container:
  - Transient container type defines transient object references for components.
  - Persistent container type defines persistent object references for components.

Container Implementation Type

- Interaction patterns between container, POA, and CORBA services:
  - stateless: transient object references; one POA servant for multiple components.
  - conversational: transient object references; one POA servant per component.
  - durable: persistent object references; one POA servant per component.

Component Categories

- Valid combinations:

Container Implementation Type	Container Type	Primary Key	Component Categories	EJB Type
stateless	transient	No	Service	-
conversational	transient	No	Session	Session
durable	persistent	No	Process	-
durable	persistent	Yes	Entity	Entity

SIEMENS CORBA

OOP 2001 MÜNCHEN

Server Programming Environment

SIEMENS CORBA

OOP 2001 MÜNCHEN

Threading

- CORBA Components support two threading models:
  - Serialize: container prevents multiple threads from entering the component simultaneously.
  - Multithread: Component is thread-safe.

EJB supports only serialize.

SIEMENS CORBA

OOP 2001 MÜNCHEN

Servant Lifetime Management

- Servants are programming objects used to execute method requests.
- Memory for servants should be managed efficiently.
- Server programmer has different design choices:
  - Selection of container type.
  - Selection of container implementation type.
  - Selection of servant lifetime policy.
  - Implementation of callback interfaces associated with choices.

SIEMENS CORBA

OOP 2001 MÜNCHEN

Servant Lifetime

- 4 different policies available:
  - Method: component activated by container on every request and then passivated.
  - Transaction: activation on first request within transaction; passivation on transaction completion.
  - Component: activation on first request; passivation when component asks to.
  - Container: activation on first request; passivation when container decides to passivate.

Container Types and Servant Lifetime

- Possible variations:

Container Implementation Type	Container Type	Valid Servant Lifetime Policies
stateless	transient	method
conversational	transient	method, transaction, component, container
durable	persistent	method, transaction, component, container

SIEMENS CORBA

OOP 2001 MÜNCHEN

Transactions

- Transaction policies:

Transaction Attribute	Client Transaction	Component Transaction
NOT_SUPPORTED	-	-
	T1	-
REQUIRED	-	T2
	T1	T1
SUPPORTS	-	-
	T1	T1
REQUIRES_NEW	-	T2
	T1	T2
MANDATORY	-	Exception
	T1	T1
NEVER	-	-
	T1	Exception

Security

- Security is consistently applied to all component categories.
- Container extracts policy from deployment descriptor.
- It checks active credentials and adjusts them to accomodate requested policy.
- Policy remains unchanged until change request.

Events

- CORBA Components use subset of notification service.
- Event data structure mapped to an any.
- Events are passed as valuetypes.
- Push model used.
- Events may have transactional behavior depending on container implementation type and deployment descriptor.
- Channel management by container.
- Possible policies *normal*, *default*, *transaction*

Persistence

- A persistent container type supports use of persistence mechanism:
  - Container-managed persistence: component developer defines state. Container saves and stores state automatically.
  - Component-managed persistence: component in charge of saveing and storing state.
  - It is possible to use the PSS or a user-defined service.

Operation Invocation

- Components provide (multiple) supported and provided interfaces.
- Operations may raise exceptions.
- User exceptions are forwarded to clients directly and do not affect transactions.
- Other exceptions intercepted by container and lead to rollback (exception).

SIEMENS CORBA

OOP 2001 MÜNCHEN

Interfaces between components and containers

- There are two kinds of interfaces:
  - Internal interfaces are provided by the container and called by the component.
  - Callback interfaces are provided by the component and called by the container.

Interfaces for all Container Types

- ComponentContext is an external interface to access runtime services:

```
exception IllegalState {};
local ComponentContext {
    // get reference used to invoke component:
    CORBA::Object get_reference() raises (IllegalState);
    HOMEBase get_home(); // get reference to home
    Transaction get_transaction(); // get a Transaction interface
    HomeRegistration get_home_registration(); // get reference to HR
    Security get_security(); // get Security interface
    Events get_events(); // get Events interface
};
```

Internal Interfaces

- Home: find/locate components
- BaseOrigin interface: used by component to request passivation:

```
exception PolicyMismatch {};
local BaseOrigin {
     void req_passivate() raises (PolicyMismatch);
};
```

Internal Interfaces (cont'd)

- Transaction Interface:

```
exception NoTransaction {}; exception InvalidCookie {};
enum Status {ACTIVE, MARKED_ROLLBACK, PREPARED, COMMITTED, ROLLED_BACK,
NO_TRANSACTION, PREPARING, COMMITTING, ROLLING_BACK};
local Transaction {
      void begin();
      void commit() raises(NoTransaction);
      void rollback() raises(NoTransaction);
      void set_rollback_only() raises(NoTransaction);
      boolean get_rollback_only() raises(NoTransaction);
      LocalCookie suspend() raises(NoTransaction);
      void resume(in LocalCookie cookie) raises(InvalidCookie);
      Status get_status();
      void set_timeout(in long to);
};
```

Internal Interfaces (cont'd)

- HomeRegistration Interface: internal interface used by component to register its home so it can be located.
- Security Interface:

```
typedef SecurityLevel2::Credentials  Principal;
local Security {
     Principal get_caller_identity();
     boolean is_caller_in_role(in Principal role);
};
```

Internal Interfaces (cont'd)

- Events interface: external interface to support emitting and publishing events.
- Components must implement a callback interface derived from:

```
local EnterpriseComponent {
};
```

SIEMENS
CORBA

OOP 2001 MÜNCHEN

Client Programming Model

Client/Component Interaction

- The client interacts through two forms of external interfaces:
  - One or more application interfaces
  - Home interface
- The home supports two patterns:
  - If primaryKey is defined: factories and finders
  - If primaryKey is not defined: factories

Component-aware clients

- Clients that know that they deal with components are component-aware.
- They know how to use all of the interfaces and the home interface.
- They locate these interfaces using Components::HomeFinder or the naming service.
- The starting point is `resolve_initial_references()`.
- Use „ComponentHomeFinder“ to locate home finder.

Same example again

```
// Get home finder:
org.omg.CORBA.Object objref =
   orb.resolve_initial_references(„ComponentHomeFinder“);
// „cast“ it to correct type:
ComponentHomeFinder hf = ComponentHomeFinderHelper.narrow(objref);
// find home using type:
org.omg.CORBA.Object of = hf.find_home_by_type(BankHomeHelper.id());
// „cast“ it to correct type:
BankHome bh = BankHomeHelper.narrow(of);
// create instance and narrow it:
org.omg.Components.ComponentBase bankInstance = bh.create();
Bank myBank = BankHelper.narrow(bankInstance);
// invoke operations:
CORBA::ULong how_much_money = myBank.amount();
```

Component-unaware clients

- Clients that do not know they deal with a component.
- These clients only see one interface, namely the supported interface of a component.
- They obtain initial references to the home by using the name service or trader service.
- After creation of the component, they use its standard interface.

SIEMENS CORBA

OOP 2001 MÜNCHEN

Persistence Concepts

- The CIF defines:
  - A storage type is an abstract state of an executor, managed by the component or the framework.
  - Storage objects are instances of storage types. They are managed by a object store and created through a storage home.
  - An incarnation is a programming interface that manifests a storage object in an execution context. It hides all the system details.

Persistence Concepts (cont'd)

- A storage home defines an interface to manage a specified storage type, find incarnations, create storage objects and destroy instances.
- Persistent stores are the primary point of contact between the application and the storage mechanisms. They maintain the state of storage objects, and the ACID attributes. They also provide storage homes.
- Persistent Ids (PIDs) are values that uniquely identify storage objects within a persistent store.
- Primary keys can be optionally defined in storage homes.

Example

```
storage Account {
    long account_number;
    long amount;
};
storage Customer {
    string owner;
    Account account;
};

... IS MAPPED TO Java:
```

Example (cont'd)

```
interface AccountAbstractState {
     long account_number();
     void account_number(long val);
     long amount();
     void amount(long val);
}
interface Account extends AccountAbstractState,
IncarnationBase {}
interface CustomerAbstractState {
     string owner();
     void owner(owner val);
     Account  account();
     void account(Account val);
}
```

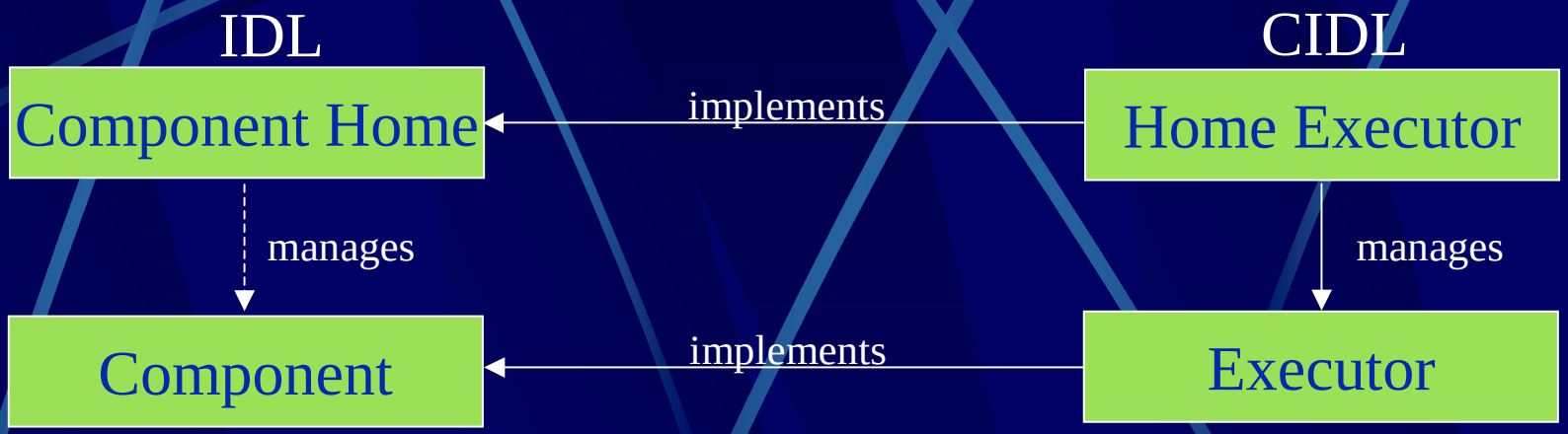
Example (cont‘d)

```
Customer customer = CustomerHome.create();
// account member in customer implicitely
// created.
customer.account().account_number(1234);
customer.account().amount(0);
customer.owner(„Michael Stal“);
```

Composition

- The composition denotes the collection of all necessary artifacts of an implementation:
  - Home / home executor
  - Component / component executor
  - Abstract storage home
  - storage

Simple Composition




```
composition entity MyComponent {
        home executor MyHomeExecutorName
        implements MyHome
        manages MyComponentExecutor
}
```

Packaging and Deployment

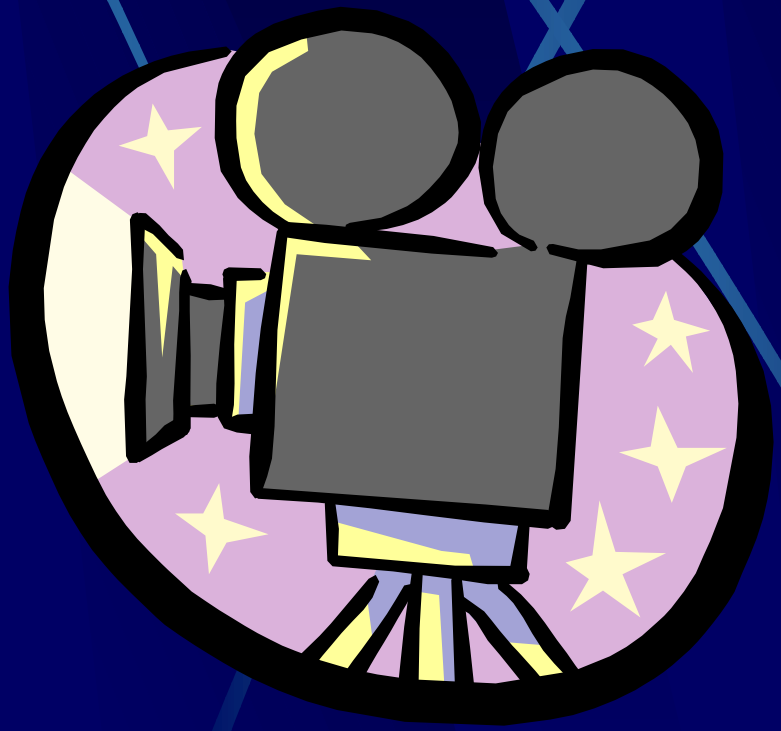
Packaging

- A CORBA Component Package represents one or more implementations of an abstract component.
- It may be installed or grouped together with other components to form an assembly.
- A package consists of a descriptor and a set of files.
- All files of a package are either part of an archive file or stored separately.
- In the last case the descriptor points to the file locations.
- Descriptors are XML documents.

SIEMENS
CORBA

OOP 2001 MÜNCHEN

EJB Mapping Issues



CCM and EJB

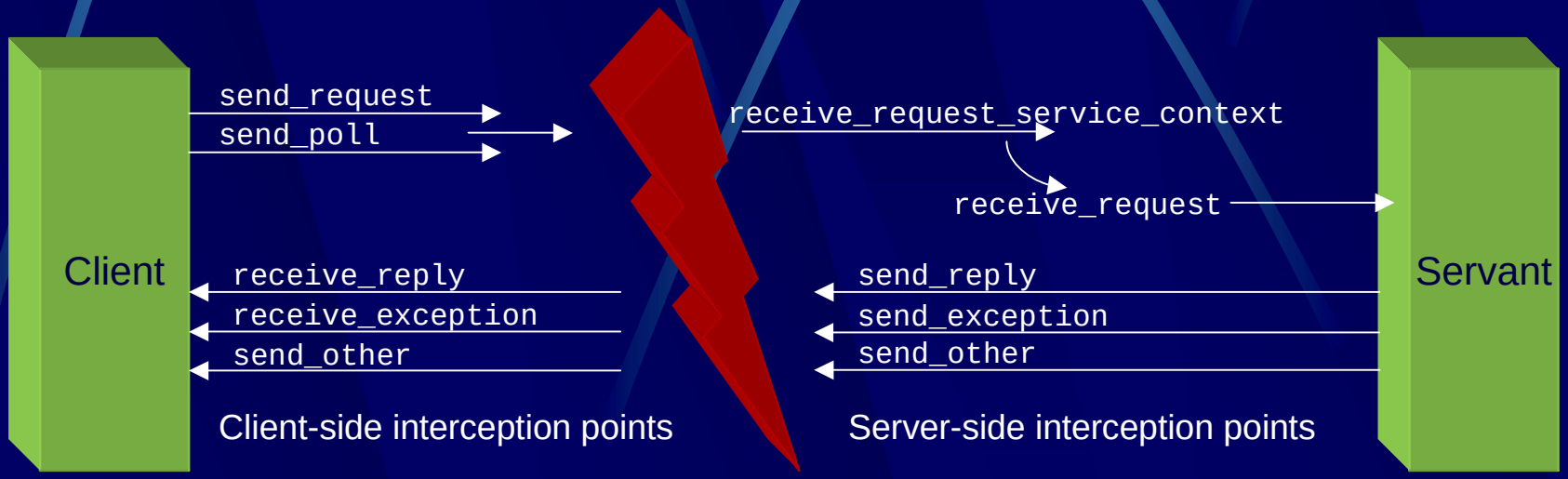
- CORBA Components strives for compatibility with Enterprise JavaBeans:
  - (1) CORBA Components can be used by Java clients; EJBs can be used by CORBA Clients.
  - (2) CORBA Component containers can support EJBs.
  - (3) CORBA components written in Java that follow the EJB patterns are deployable in EJB containers:
    - Either the component is an EJB; or
    - The component has two different faces: that of an EJB and that of a CORBA component.

Portable Interceptor Framework

- Interceptors are used to add out-of-the-band functionality during ORB processing.
- Previously, this functionality could either not be integrated into existing ORBs or only in a product specific way.
- Portable Interceptors solve this problem.

Example: Client-Side Interceptor

- Let us consider an example. A client sends a request to a server:



Example (cont'd)

- For a specific kind of ORB processing (e.g., client requests) points in the event flow (interception points) are identified where an interceptor should be able to intercept.
- An interceptor is registered with the ORB. For each interception point a separate method is available in the interceptor interface.
- Interception points might be interrelated by the control flow. There might be starting points, intermediate points, ending points.

Example (cont'd)

- In a client request there are different interception points:
  - send_request: interceptor might query information on request, raise a system exception.
  - send_poll: TII polling request. May raise system exception.
  - receive_exception: query exception before exception is raised to client. May raise a system exception or a ForwardRequest exception.
  - receive_other: allows to query information when request is something other than normal reply or exception. For example a request colld result in a retry. May raise system exception.

Example (cont'd)

```
module PortableInterceptor {
   local interface Interceptor {
        readonly attribute string name;
   }
}

local interface ClientRequestInterceptor : Interceptor {
   void send_request(in ClientRequestInfo ri) raises
   (ForwardRequest);
   void send_poll(in ClientRequestInfo ri); // TII
   void receive_reply(in ClientRequestInfo ri);
   void receive_exception(in ClientRequestInfo ri) raises
   (ForwardRequest);
   void receive_other(in ClientRequestInfo ri) raises
   (ForwardRequest);
};
```

Example (cont'd)

- More than one interceptor can be registered with an ORB for the same event.
- Interceptors are logically put on a virtual stack one after another and then one of the starting points gets called.
- An ending point is only called for all interceptors pushed on the virtual stack.

Registration of Interceptors

- Example:

```
public class LoggingService implements ORBInitializer {
   // pre_init is called during ORB initialization.
   void pre_init(ORBInitInfo info) {
        Interceptor interceptor = new LoggingInterceptor;
        info.add_client_request_interceptor(interceptor);
   }
   // all initial references are available
   void post_init(ORBInitInfo info) {
        // ...
   }
}
```

Java: set initializer class with –D option at program start

Summary

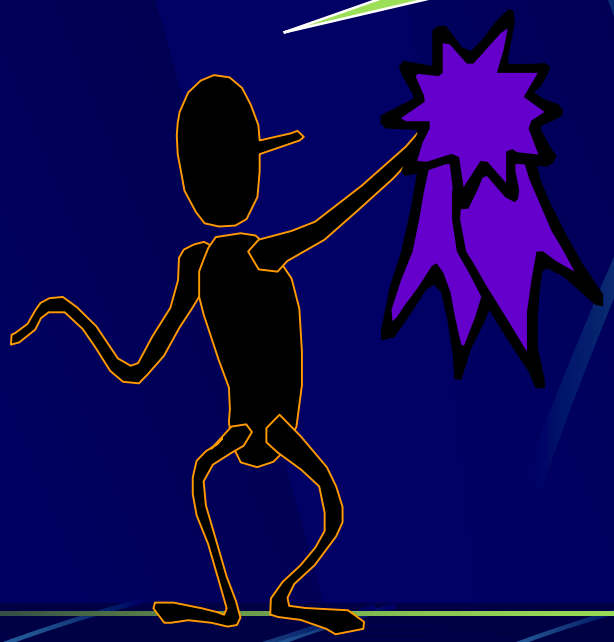
- Famous last words

Summary

- CORBA Components is an easy-to-use, powerful technology for building platform-independent middle-tier components.
- Enterprise JavaBeans and CORBA Components are two sides of the same coin.
- With CORBA 3, the OMG offers a full range of enterprise technologies.
- What we need now are products, products, ...

SIEMENS
CORBA
OOP 2001 MÜNCHEN

Thanks a lot for attending this talk!!

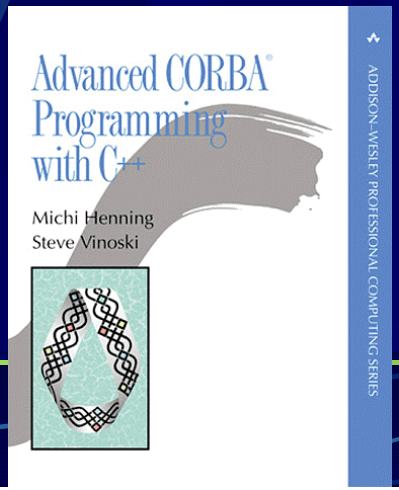


SIEMENS
CORBA
OOP 2001 MÜNCHEN

Any Questions ?

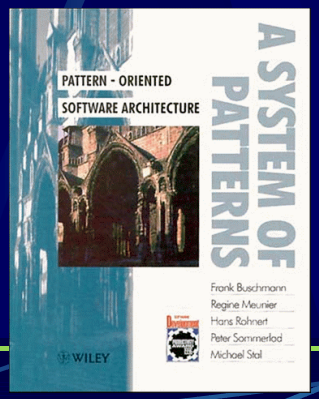
CORBA Book References

- Henning, Vinoski: Advanced CORBA Programming with C++, Addison Wesley, 1999. (The „bible“ for CORBA programmers).
- Puder, Römer: Middleware für verteilte Systeme, dpunkt, 2000.
- Ruh, Herron, Klinker: IIOP Complete, Wiley, 1999.
- Pritchard: COM and CORBA Side by Side: Architectures, Strategies, and Implementations, 1999, Addison-Wesley.
- Siegel (ed.): CORBA 3 Fundamentals and Programming, 2nd Edition, 2000, Wiley.



Patterns References

- Buschmann, Meunier, Rohnert, Sommerlad, Stal: Pattern-Oriented Software Architecture - A System of Patterns, Wiley, 1996.
- Schmidt, Stal, Rohnert, Buschmann: Pattern-Oriented Software Architecture Vol. 2 – Patterns for Concurrent and Networked Objects, Wiley, 2000.
- Gamma, Helm, Johnson, Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.



Internet References

- OMG http://www.omg.org
- BEA: http://www.beasys.comh
- IBM: http://www-4.ibm.com/software/ad/cb/
- Inprise/Borland: http://www.borland.com
- IONA: http://www.iona.com
- Bean Homepage: http://java.sun.com/products/ejb
- The ACE ORB http://www.cs.wustl.edu/~schmidt

Additional Bonus Material

Standardized CORBA Services

- *Naming*: creation of name spaces and translation of names to object references.
- *Lifecycle*: creation, modification, copying, movement and removal of objects.
- *Events*: asynchronous messaging.
- *Persistence*: persistent store and retrieval of objects.
- *Concurrency*: parallel access to objects by standard mechanism like locks and semaphores.
- *Externalization*: export of objects to system external files.
- *Relationship*: object relations (f.i. 1:n, n:m).
- *Transaction*: transaction oriented access with 2 level commit.

Standardized CORBA Services (cont'd)

- *Licensing*: framework for specification and management of license servers.
- *Query*: predicate based and declarative operations on collections of objects.
- *Time*: synchronization of clocks in distributed environments.
- *Security*: authorizing and supervising at object level.
- *Properties*: typed and attributed values, statically or dynamically attached to an object.
- *Trading*: Clients can ask for services and specify properties. They do not care about servers.
- *Collections*: Collections a la CORBA.

Event Service Example

Supplier:

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
CORBA::Object_var obj = orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var inc;
inc = CosNaming::NamingContext::_narrow(obj);
assert(!CORBA::is_nil(inc.in()));
CosNaming::Name ec_name;
ec_name.length(1);
ec_name[0].id = CORBA::string_dup("CosEventService");
obj = inc->resolve(ec_name);
if (CORBA::is_nil(obj.in())) {
          cerr << "Could not find Event Service" << endl;
          return 1;
}
CosEventChannelAdmin::EventChannel_var echoEC =
          CosEventChannelAdmin::EventChannel::_narrow(obj);
if (CORBA::is_nil(echoEC.in())) {
          cerr << "Invalid reference for event channel" << endl;
          return 1;
} // .. to be continued
```

Event Service Example (cont'd)

Sample supplier (continued):

```
// continued ...
CosEventChannelAdmin::SupplierAdmin_var supplierAdmin =
echoEC->for_suppliers();
CosEventChannelAdmin::ProxyPushConsumer_var consumer =
            supplierAdmin->obtain_push_consumer();
consumer->connect_push_supplier(CosEventComm::PushSupplier::_nil());
// Now, we can fire an event:
CORBA::Any event;
event <<= CORBA::string_dup("insertEntry");
consumer->push(event);
```

Event Service Example (cont'd)

Sample consumer (consumer declaration):

```
#include <orbsvcs/CosEventCommS.h>
class HashConsumer_i : public virtual POA_CosEventComm::PushConsumer
{
public:
HashConsumer_i(CORBA::ORB_ptr orb);
virtual void push(const CORBA::Any & data,
     CORBA::Environment &ACE_TRY_ENV = CORBA::default_environment());
virtual void disconnect_push_consumer(
     CORBA::Environment &ACE_TRY_ENV = CORBA::default_environment());
private:
     CORBA::ORB_var orb_;
};
```

Event Service Example (cont'd)

Sample consumer (consumer definition):

```
#include "hashconsumer_i.h"
#include <iostream.h>
HashConsumer_i::HashConsumer_i(CORBA::ORB_ptr orb) : orb_(CORBA::ORB::_duplicate(orb)){}

void HashConsumer_i::push(const CORBA::Any &data, CORBA::Environment &ACE_TRY_ENV) {
      char *eventString;
      if (data >>= eventString) {
                cout << "Got event : " << eventString << endl;
      }
}
void HashConsumer_i::disconnect_push_consumer(CORBA::Environment &ACE_TRY_ENV) {
      CORBA::Object_var obj = orb_->resolve_initial_references("POACurrent");
      PortableServer::Current_var current = PortableServer::Current::_narrow(obj);
      PortableServer::POA_var poa = current->get_POA();
      PortableServer::ObjectId_var objectId = current->get_object_id();
      poa->deactivate_object(objectId);
}
```

Event Service Example (cont'd)

Sample consumer (main):

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
CORBA::Object_var obj = orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var inc;
inc = CosNaming::NamingContext::_narrow(obj);
assert(!CORBA::is_nil(inc.in()));
CosNaming::Name ec_name; ec_name.length(1);
ec_name[0].id = CORBA::string_dup("CosEventService");
obj = inc->resolve(ec_name);
if (CORBA::is_nil(obj.in())) {
        cerr << "Could not find Event Service" << endl;
        return 1;
}
CosEventChannelAdmin::EventChannel_var echoEC = CosEventChannelAdmin::EventChannel::_narrow(obj.in());
if (CORBA::is_nil(echoEC.in())) {
        return 1;
} / / continued
```

Event Service Example (cont'd)

Sample Consumer (main cont'd)

```
HashConsumer_i servant(orb.in());
obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
CosEventComm::PushConsumer_var consumer = servant._this();
CosEventChannelAdmin::ConsumerAdmin_var consumerAdmin = echoEC->for_consumers();
CosEventChannelAdmin::ProxyPushSupplier_var supplier =
                                        consumerAdmin->obtain_push_supplier();
supplier->connect_push_consumer(consumer.in());

PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate();

orb->run();
```

SIEMENS CORBA

OOP 2001 MÜNCHEN

Lifecycle Service

- Sometimes, multiple copies of a CORBA type are necessary with clients capable of controlling lifecycle and location.
- The Lifecycle Service is more a guideline than an implementation.
- A Lifecycle object can be moved, removed, or copied.
- For each Lifecycle type a factory is provided that creates instances of the type.
- Factory finders denote the location where factories and their objects live.

Lifecycle Service (cont'd)

CORBA class that supports LifeCycle objects must derive from:

```
interface LifeCycleObject {
 LifeCycleObject copy(in FactoryFinder there, in Criteria the_criteria)
     raises(NoFactory, NotCopyable, InvalidCriteria, CannotMeetCriteria);
 void move(in FactoryFinder there, in Criteria the_criteria)
     raises(NoFactory, NotMovable, InvalidCriteria, CannotMeetCriteria);
 void remove() raises(NotRemovable);
};
```

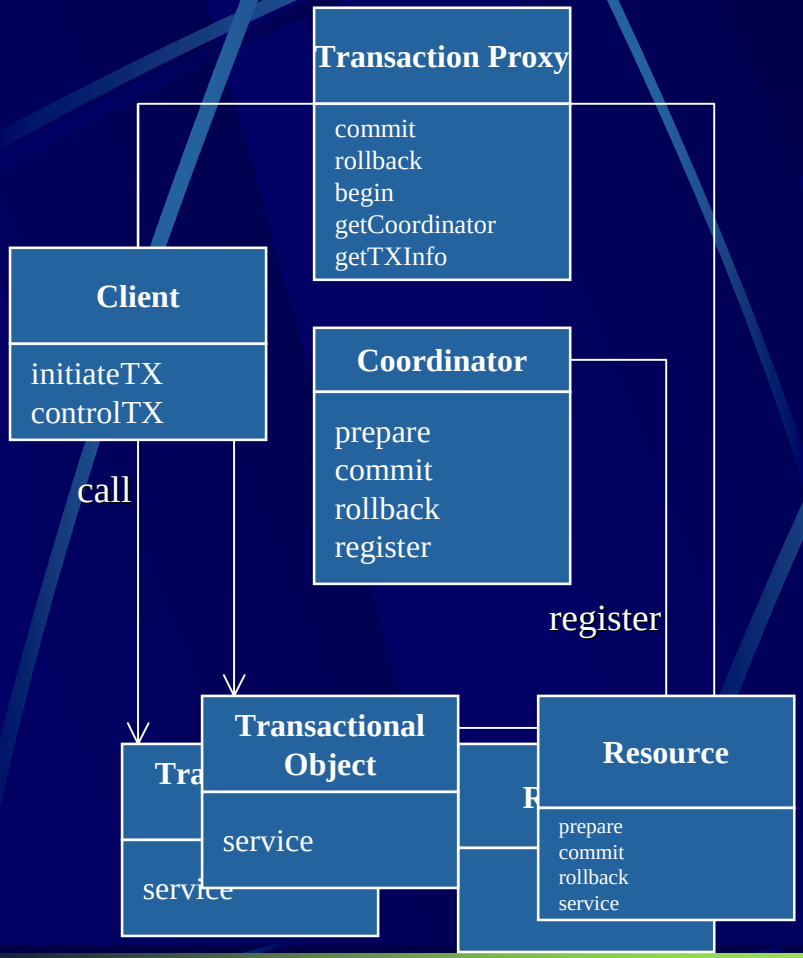
Factories themselves are diverse. Nonetheless, a generic factory interface is provided by the specification. Factory Finders locate factories:

```
interface FactoryFinder {
      Factories find_factories(in Key factory_key) raises (NoFactory);
};
```

CORBA Time Service

- The CORBA Time Service helps to synchronize time in a distributed environment.
- Time based on UTC (*Universal Time Coordinated*) which specifies the units of 1/10 msecs elapsed since Oct. 15, 1582 (Gregorian Calendar).
- UTO (*Universal Time Object*) specifies a relative or absolute time and an inaccuracy value.
- TIO (*Time Interval Objects*) specifies time intervals.
- The *TimerEventService* allows to create TimerEventHandlers.
- A *TimerEventHandler* triggers time events using the Push-Style Event Service.

Transaction Service



A transaction-based paradigm for distributed objects:

A *Client* is responsible for defining transaction boundaries as well as for starting and stopping transaction.

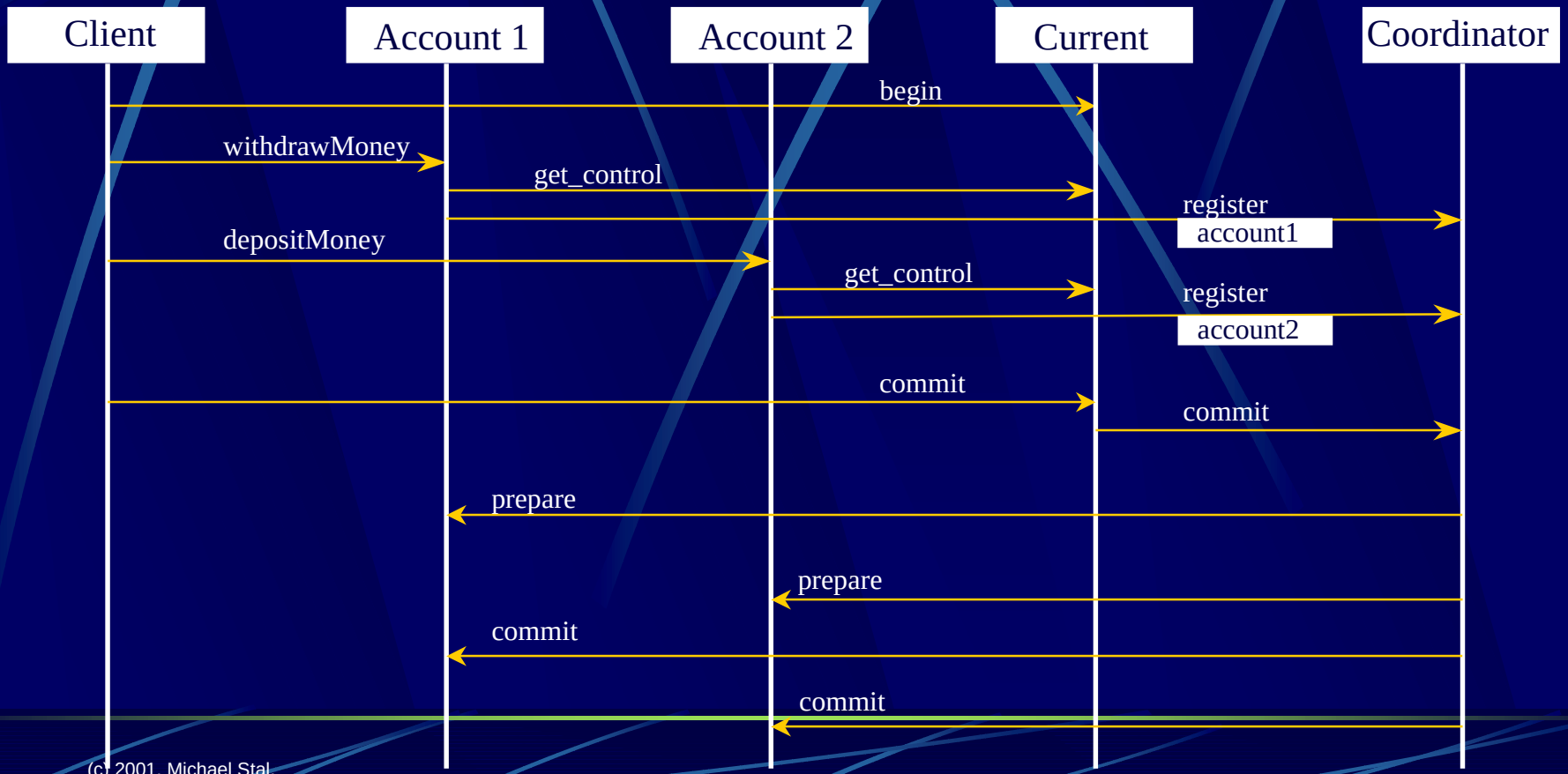
Coordinators coordinate the cooperation of distributed objects within a transaction.

A *Resource* changes its internal state during a transaction. It registers with the coordinator.

A Transactional Object participates in a transaction but its state remains unaffected by the transaction.

A *Transaction Proxy* allows clients and objects to retrieve information and control transactions.

Transaction Service (cont'd)



Transaction Service (cont'd)

- **The OTS reveals the following features:**
  - **OTS transactions can be combined with X/OPEN DTP procedural transactions.**
  - **One ORB can support multiple transaction services. Transactions might span multiple ORBs.**
  - **To use transactions in your server objects, just inherit them from the abstract OTS interface.**
  - **Support for flat transactions mandatory, nested transactions are optional.**

SIEMENS CORBA

OOP 2001 MÜNCHEN

Memory Management

- **Note that CORBA is location transparent. Thus, memory is allocated as if everything were local. This is trivial in Java because Java uses Garbage Collection, but more complex in C++:**
- *in* parameters: caller allocates and deallocates memory. If you are assigning the value of an *in* parameter to a local variable of a callee, you must duplicate the value: `MyInterface::_duplicate(inarg)`.
- *out* parameters: callee allocates memory, caller deallocates memory.
- *inout* parameters: caller allocates memory, callee deallocates and reallocates memory, caller deallocates reallocated memory.

Memory Management (cont'd)

- Since string allocation and deallocation is machine-dependent use methods such as `CORBA::string_dup()`, `...`
- In C++ array types can not be passed by value. Thus the definition: `typedef T Tarr[7]` becomes: `typedef T Tarr[7]` and `typedef T* Tarr_slice`. in-parameters and return values are then passed as `Tarr_slice`.
- For releasing references use `CORBA::release()`.
- For not simple types `T` the C++ mapping generates `T_ptr` and `T_var` types. `T_var` as a smart pointer class automatically deallocates memory when necessary. There is also a `String_var` type.

Memory Management (cont'd)

- There is one problem left. What about out-parameters. Consider, for instance, `f(out string x)`. If you call this function two times and pass a `char *`, or a `String_var` the called should overwrite `x` in the first scenario but properly deallocate and reallocate `x` in the second scenario.
- For this purpose, CORBA introduces `_out` types. Example: `String_out`. This type automatically detects the type of the *in*-parameter and behaves properly in both cases.

Memory Management (cont'd)

- In Java simple types such as `long` are always passed by value, complex types are passed by reference.
- What if you want to use a `long`-out or an object reference as an out-parameter? For this purpose, Java-IDL generates `Holder`-classes.
- Thus, inout/out-parameters are passed as Holder classes:

```
interface T { ... };
// Among other things the compiler generates:
public class THolder {
        private T value_;
        public void set_value(T val) { value_ = val; }
        public T get_value() { return value_; }
}
```