Problems and Solutions in Classical Component Systems

- Language Transparency
- Location/Distribution Transparency
- Example: Yellow Page Service
- IDL principle
- Reflective Calls, Name Service

Remember: Motivation for COTS

Component definition revisited:

- **Program units for composition** with
  - *standardized* basic communication
  - *standardized* contracts
  - *independent* development and deployment
- A meaningful **unit of reuse**
  - Large program unit
  - Dedicated to the solution of a problem
  - Standardized in a likewise standardized domain
- Goal: economically stable and **scalable** software production

Obstacles to Overcome …

- **Technical – Interoperability**
  - Standard basic communication
  - Heterogeneity:
    different platforms, different programming languages
  - Distribution:
    applications running on locally different hosts
    connected with different networks
- **Economically – Marketplace**
  - Standardize the domain
    to create reusable, standardized components in it
  - Create a market for those components
    (to find, sell and buy them)
    – which has some more technical implications

Technical Motivations

When the Object Management Group (OMG) was formed in 1989, **interoperability** was its founders' primary, and almost their sole, objective:

A vision of software components working smoothly together, without regard to details of any component's location, platform, operating system, programming language, or network hardware and software.

- Jon Siegel

Interoperability problems to be solved by component systems

- ***Language transparency***:
  interoperability of programs
  - on the same platform, using
  - different programming languages
- ***Platform transparency***:
  interoperability of programs
  - written for different platforms using
  - the same programming language
- ***Heterogeneity***:
  - Different platforms, different programming languages
  - Requires language and platform transparency

Language Transparency Problems

- Calling concept
  - Procedure, Co-routine, Messages, …
- Calling conventions and calling implementation
  - Call by name, call by value, call by reference, …
  - Calling implementation: Arguments on stack, in registers, on heap, ...
- Data types
  - Value and reference objects
  - Arrays, unions, enumerations, classes, (variant) records, …
- Data representation
  - Coding, size, little or big endian, …
  - Layout of composite data
- Runtime environment
  - Memory management, garbage collection, lifetime …

Options In General

For n languages:

- Direct language mapping:
  - 1:1 adaptation of pairs of languages: $O(n^2)$
- Mapping to common language:
  - Adaptation to a general exchange format: $O(n)$
- Compiling to common type system:
  - Standardize a single format (as in .NET): $O(1)$ but very restrictive, because the languages become very similar

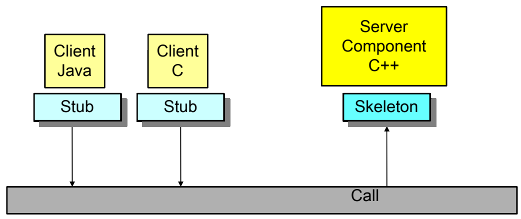
Solutions in Classical Component Systems

- Calling concept:
  - standardized by the communication library (RPC)
- Calling conventions and implementation:
  - Standardized by the communication library (EJB - Java , DCOM - C)
  - Implementation for every single language (CORBA)
- Data types:
  - Existing type system as standard (EJB – Java types)
  - New standard type system (CORBA IDL-to-Language mapping)
- Data representation:
  - Standard (EJB – Java representation, DCOM – binary standard)
  - Adaptation to a general exchange format (CORBA GIOP/IIOP)
- Runtime environment
  - Standard by services of the component systems

Language Transparency Implementation

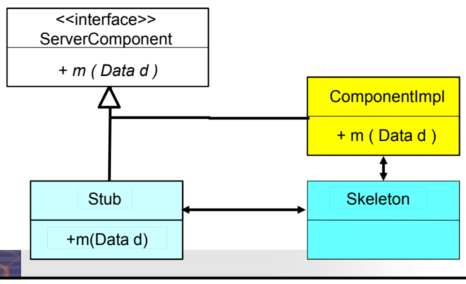
- Stubs and Skeletons
  - ***Stub***
    - Client-side proxy of the component
    - Takes calls of component clients in language A and sends them to the
  - ***Skeleton***
    - Takes those calls and sends them to the server component implementation in language B
- Language adaptation could take place in Stub or Skeleton (or both)
  - Adaptation deals with calling concepts, data formats, etc.
- Solution of distribution transparency problem postponed ...

Stubs and Skeletons



Stubs and Skeletons

- A typical instance of the **proxy pattern**
  - Stub (client-side proxy) delegates calls to Skeleton
  - Skeleton (server-side proxy) delegates to servant (implementation)



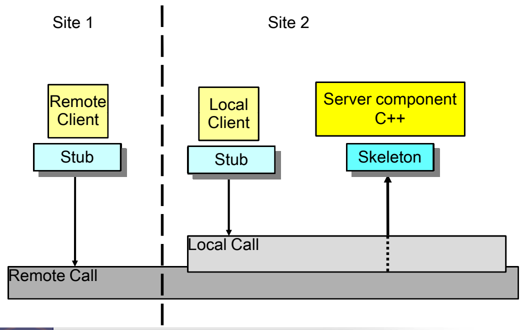
Distribution

- ***Location transparency / Distribution transparency:*** interoperability of programs independently of their execution location
- Problems to solve:
  - Transparent basic communication
    - Transparently initiate a local/remote call
    - Transparently transport data locally or remotely via a network
    - How to handle references transparently?
  - Distributed systems are heterogeneous
    - So far, we handled platform-transparent design of components
  - Usual suspects in distributed systems
    - Transactions
    - Synchronization
    - …

Transparent Local/Remote Calls

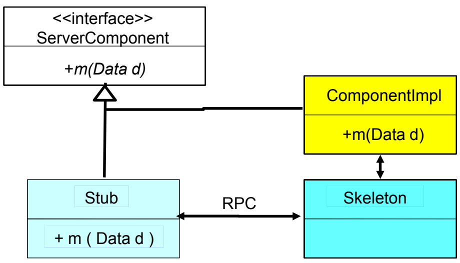
- Communication over proxies (-> proxy pattern)
  - Proxies redirect call locally or remotely on demand
  - Proxies always local to the caller
- RPC for remote calls to a handler
  - Handler always local to the callee
- Déjà vu! We reuse Stubs and Skeletons

Remote Stubs and Skeletons

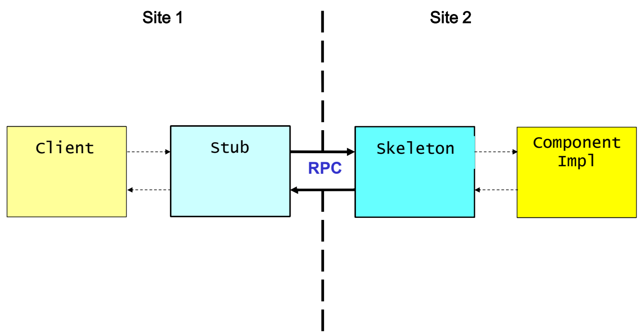


Stubs and Skeletons for Distribution

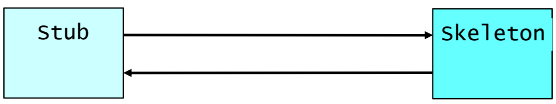
- A variant of the Proxy pattern,
  using remote procedure call (RPC) when forwarding requests



Stubs and Skeletons



Stubs and Skeletons so far …
(same platform)



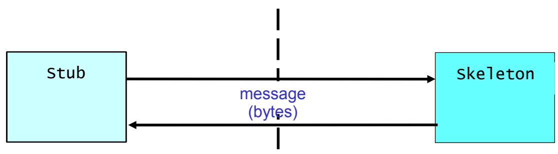
Language 1

1. Map call data to an exchange format
2. Call Skeleton

Language 2

3. Receive Call from Stub
4. Retrieve data from the exchange format

… and now



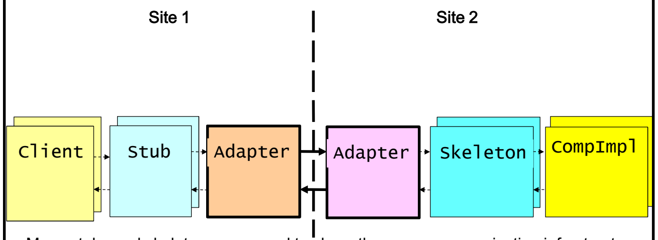
Language 1
Site 1

1. Map data / call to a byte stream exchange format
2. Send message, e.g. via TCP/IP network socket

Language 2
Site 2

3. Receive message from network socket
4. Retrieving data / call from the byte stream exchange format

Stubs, Skeletons, and Adapters



Many stubs and skeletons may need to share the same communication infrastructure (e.g., TCP/IP ports)

Stub and skeleton objects must be created and referenced by need.

Put this support functionality in a separate **Adapter** layer ("run-time system for RPC")

Remark: In CORBA, this "Adapter" functionality will be split between the ORB (communication) and the so-called Object-Adapter (multiplexing).

TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Reference Problem

- Target of calls
- Call-by-reference parameters, references as results
- Reference data in composite parameters and results
- Scope of references
  - Thread/process
  - Computer
  - Agreed between communication partners
  - Net wide
- How to handle references transparently?

TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Approach

- World-wide unique addresses
  - E.g. computer address + local address
  - URL, URI (uniform resource identifiers)
- Mapping tables for local references
  - Logical-to-physical
  - Consistent change of local references possible
- (In principle) one adapter per computer manages references
  - 1:n relation adapter to skeletons
  - 1:m relation skeletons to component objects
  - Lifecycle and garbage collection management
  - Identification ("Who is this guy …?")
  - Authorization ("Is he allowed to do this …?")

TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

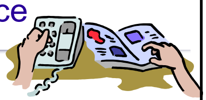
Change of Local References

Why are you interested in a reference?

- Need a reference to computation service (function)
  - Sufficient to have a reference to the component
  - Adapter creates or hands out reference to an arbitrary object on demand
- Need a reference to store/retrieve data service
  - Use a data base
  - Adapter creates or hands out an arbitrary object instance wrapping the accesses to the data base
- Need a reference to stated transaction to leave and resume
  - Adapter must keep correct the mapping logical-to-physical address
  - Problems with use of self reference inside and outside service

TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Example: Yellow-Page Service



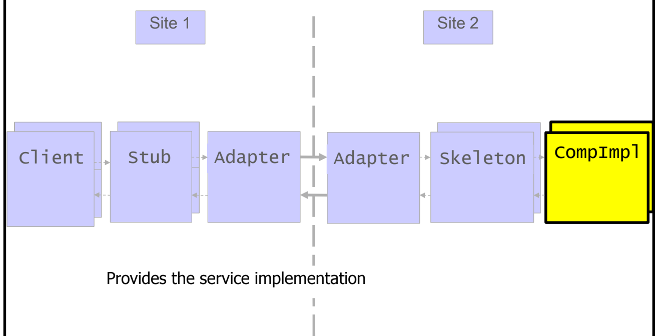
- **Yellow Pages service**
  - Lookup of a name (database access with caching by YP object)
- Internally: **2 types of requests** (in adapter/stub/skeleton layers)
  - **Lookup Request**: given
    - Service type (Yellow pages, phone book, ...)
    - Address: specifies the YP service object (i.e., a reference)
    - Requested method (lookup, ...)
    - and array of parameter objects, e.g. name (string) to look up
  - **Creation Request**: Creation of a new YP service object on server
    - Service type
    - Address = -1 (denotes creation request)

YP service objects registered in YP skeleton in a hashtable of YP objects

TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Example: Yellow Page Service (1)

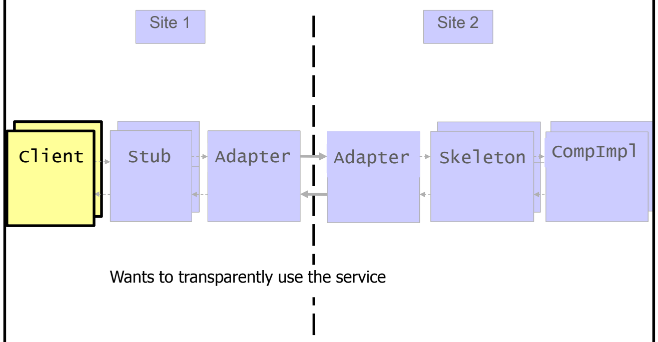
Service component



Example: Yellow Page Service (1)
Service component

```
class YellowPages extends YellowPagesInterface {
    private Hashtable cache = new Hashtable ();
    //JDBC data base connection:
    private static DataBase db = … ;
    public String lookup(String name) {
      String res;
      if ((res = cache.lookup(name)) != null)
           return (String)res;
      if ((res = db.lookup(name)) != null){
           cache.put(name,res);
           return (String)res;
      }
      return "Sorry";
    }
}
```

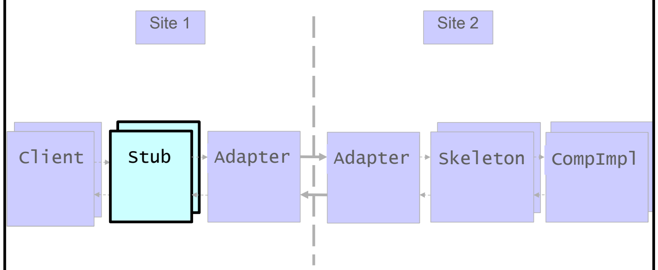

Example: Yellow Page Service (2)
Client



Example: Yellow Page Service (2)
Client

```
class Client {
    …
    YellowPageInterface yps = YellowPageInterface.getOne();
    …
    String res = (String)yps.lookup( ...string to lookup... );
    …
}
```


Example: Yellow Page Service (3)
Stub (client site)



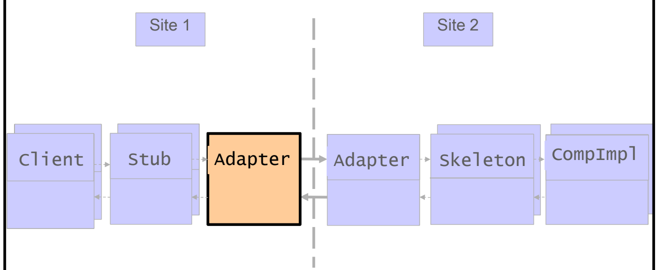
1:1 mapping to service component
Manages services objects of that component on client site
Is called from the client

Example: Yellow Page Service (3)
Client Stub

```
class YellowPageStub extends YellowPageInterface {
    private ClientAdapter ca = new ClientAdapter();
    private static Hashtable yellowPageObjects = new Hashtable();

    public String lookup(String name) {
      ca.invoke(    "Yellow Pages", yellowPageObjects.get(this),
                    "lookup", Object[]{name});
      return (String)ca.res;
    }
    // client-side constructor:
    public YellowPageInterface getOne() {
      ca.invoke("Yellow Pages", Integer(-1), "new", null);
      yp = new YellowPageStub();
      yellowPageObjects.put( yp, ca.res );
      return yp;
    }
}
```


Example: Yellow Page Service (4)
Client Site Adapter



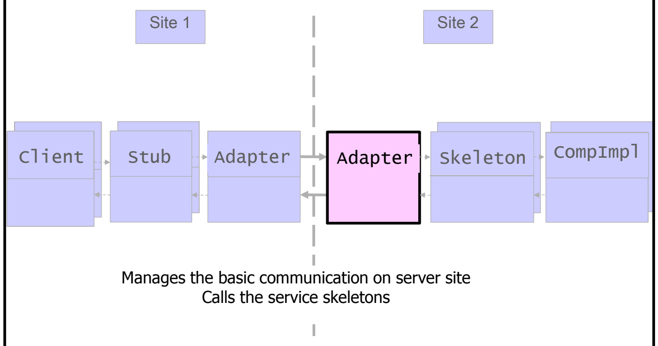
Manages the basic communication on client site
Is called from the client stubs

Example: Yellow Page Service (4)
Client Adapter

```
class ClientAdapter {
    Socket s = new Socket( serverHost, serverPort );    //magic
    public Object res;
    public void invoke( String service; Integer addr; String method; Object[] args)  {
      ObjectOutputStream os = new ObjectOutputStream(s.getOutputStream());
      ObjectInputStream is = new ObjectInputStream(s.getInputStream());
      os.writeObject(service);
      os.writeObject(addr);
      os.writeObject(method);
      if ( addr==Integer(-1) && method.equals("new") )  {
        os.flush();
        res = is.readObject();   }
     else {
        os.writeObject(args);
        os.flush();
        res = is.readObject();}
        s.close();    }
   }
}
```


TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Example: Yellow-Page Service (5)
Server-side Adapter



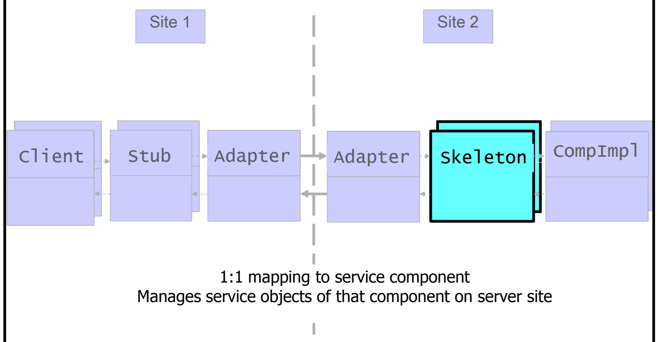
TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Example: Yellow Page Service (5)
Server-side Adapter

```
class ServiceAdapter extends Thread {
    ServerSocket ss = new ServerSocket( 0 );    //magic
    public void run() {
      while( true ) {
          try {  Socket s = ss.accept();
                    ObjectInputStream is =
                         new ObjectInputStream(s.getInputStream());
                    ObjectOutputStream os =
                         new ObjectOutputStream(s.ObjectOutputStream());
                    String service =(String) is.readObject();
                    if (service.equals("Yellow Pages")
                         new YellowPagesSkeleton(os,is).start();
                    else if (service.equals("Phone Book")
                         new PhoneBookSkeleton(os,is).start();
                    else if …
                    else System.err.println("Unknown service.");
          } catch( … ) {…}
      }
    }
}
```

TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Example: Yellow Page Service (6)
Skeleton



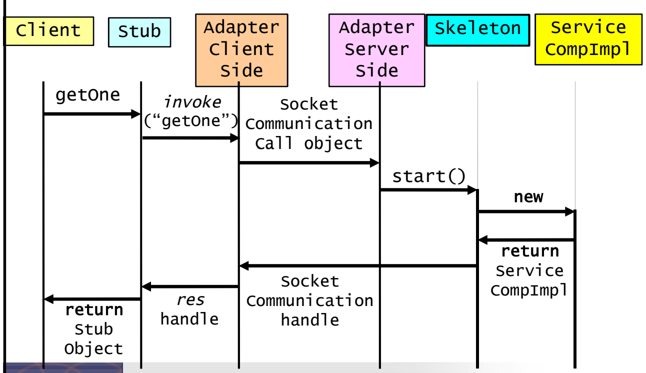
TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Example: Yellow Page Service (6)
Skeleton

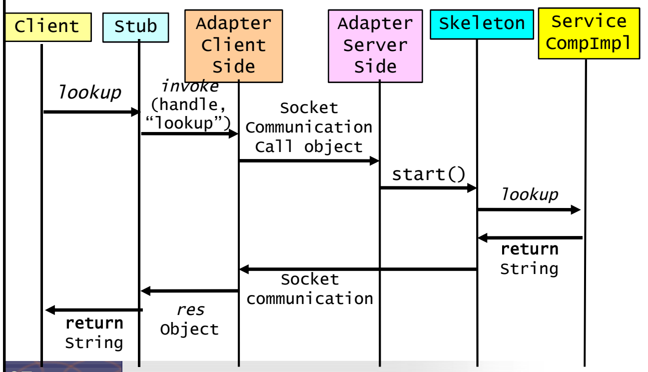
```
class YellowPagesSkeleton extends Thread implements Skeleton {
    static Hashtable yellowPageObjects = new Hashtable();
    YellowPagesSkeleton( ObjectOutputStream os, ObjectInputStream is ) { … }
  public void run() { …
     Integer addr = (Integer) is.readObject();
     if (addr == Integer(-1)) {  // creation of the service:
          Integer address = new Integer( yellowPageObjects.size() ) ;
          yellowPageObjects.put( address, new YellowPage() );
          os.writeObject( address );}
     else {  // service query:
          YellowPage yp = (YellowPage) yellowPageObjects.get( addr );
          String method = (String) is.readObject();
          if (method.equals("lookup") {
             String name = (String) is.readObject();
             String res = yp.lookup( name );  // finally: the call to the service
             os.writeObject(res); }
          else if (method.equals("store") { … }
          else System.err.println("Unknown service method."); }
     os.flush(); s.close();
   }}
```


TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Sequence Diagram, Creation



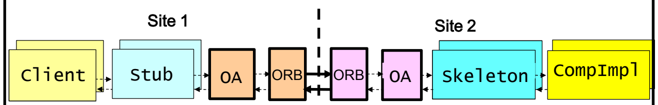
Sequence Diagram, Call



TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Technical remark

Note: **This was a simplification!**
Some issues are solved differently e.g. in CORBA or Java RMI.



"Adapter" functionality is, in CORBA, split up between ORB (communication/run-time system) and Object Adapter.

The **communication mechanism**, here Java sockets etc., is in CORBA provided by the ORB (which abstracts from language or platform specific communication mechanism/API).

The **server object registry** (static hashtable yellowPageObjects), here in the Skeleton, which is used to direct a call to the "right" server object, would in CORBA reside in the Object Adapter (who is responsible for activating / terminating "its" server processes and objects, resolving interoperable object references, and directing calls from the ORB to the right target object).

TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Who Realizes Stubs and Skeletons?

- Programmer ?
  - Much handcraft, boring and error prone
- Insight
  - Stub
    - Export interface is component dependent
    - Implementation is source language dependent
  - Skeleton
    - Import interface is component dependent
    - Implementation is target language dependent
- Idea
  - Generate export and import interfaces of Stub and Skeleton from a component interface definition
  - Take a generic language adapter for the implementation

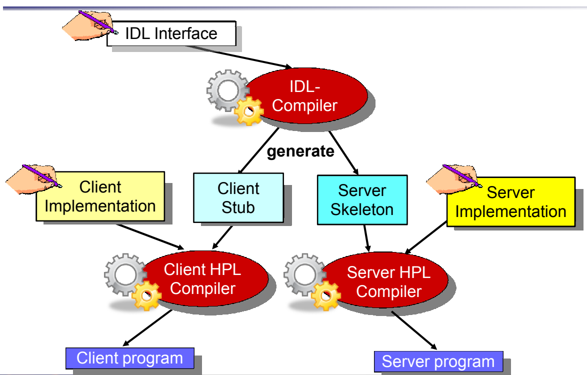
TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Interface Definition Language (IDL)

- Language to define the
  - Interfaces of components
  - Data types of parameters and results
- Programming-language independent type system
  - General enough to capture all data types in HPL (host progr. lang.)
- Procedure of construction
  - Define component with IDL
  - Generate stubs and skeletons with required languages using an IDL compiler
  - Implement the frame (component) in respective language (if possible reusing some other, predefined components)

TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Automatic Generation of Stubs and Skeletons



TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

IDL Interface Can Be Generated

Specification in IDL and host language

- Determined language binding,
- standardized IDL-to-language mapping
- Generation of stubs and skeleton is IDL-compiler independent
- Language-specific IDL compilers
- CORBA

Specification in host language only

- Retrieve the IDL spec from the HPL interface definitions (see lecture on metaprogr.)
- Have only one source of IDL compilers, guaranteeing consistency
- Quasi standard
- Java, DCOM, .NET

Required Formal Properties of the IDL-to-Language mapping

Let $\tau_{PL}: IDL \rightarrow TS_{PL}$ be the mapping from an interface definition language IDL to the type system TS of a programming language PL

- ***Well-definedness***
  for all PL : $\tau_{PL}: IDL \rightarrow TS_{PL}$ is well defined
- ***Completeness***
  for all PL : $\tau_{PL}^{-1}: TS_{PL} \rightarrow IDL$ is well defined
- ***Soundness***
  for all PL : $\tau_{PL}^{-1}\tau_{PL}: IDL \rightarrow IDL$ is ι_{IDL}
  for all PL : $\tau_{PL}\tau_{PL}^{-1}: TS_{PL} \rightarrow TS_{PL}$ is ι_{PL}

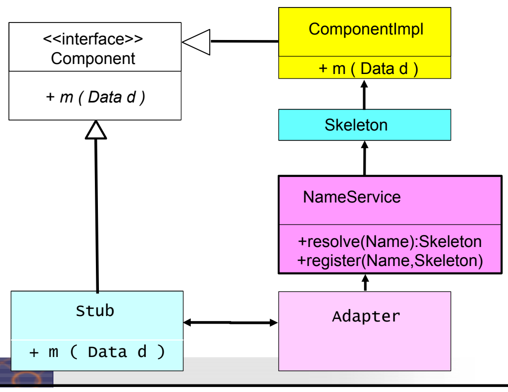
TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Example revisited

- IDL compiler must generate code for server-side adapter (example code contained the service dispatcher)
  - This is very nasty
  - One server-side adapter per site – should be independent of client components provided
  - Current solution prevents dynamic loading of services
- Idea:
  - Decoupling of adapter and skeletons
  - Provide a basic (name) service for identifying the components (skeletons) of a site
  - Components register with name and reference
  - Generic adapter provides this service

TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Name Service



TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Example: Generic Server Adapter

```
class ServiceAdapter extends Thread {
   ServerSocket ss = new ServerSocket( 0 );
   NameService ns = new NameService();
   public void run() {
     while( true ) {
      try {
        Socket s = ss.accept();
        ObjectInputStream is  =  new ObjectInputStream ( s.getInputStream() );
        ObjectOutputStream os  = new ObjectOutputStream (s.getOutputStream());
        String service = (String) is.readObject();
        Skeleton sk = null;
        if ((sk = ns.resolve(service)) != null) {
              sk.init( os, is );
              sk.start(); }
        else System.err.println("Unknown service.");
      } catch(...) {...} ...
     }
   }
```


TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Name Server Generalized

- Search for the right site providing a desired component (*extended name service*)
- Search for a component with known properties, but unknown name (*trader service*)
  - Like an extended name service
  - Components register with name, reference, and properties
  - Match properties instead of names
  - Return reference (site and service)
  - Needs standardized properties (Terminology, Ontology)
    - Functional properties (domain specific functions …)
    - Non-functional properties (quality of service …)

TDDD05 Component-based software. IDA, Linköpings universitet. C. Kessler, 2005-2013. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden.

Summary

- Component systems provide location, language and platform transparency
  - Stub, Skeleton
    - One per component
    - Technique: IDL compiler
  - Adapters on client and server site
    - Generic
    - Technique: Name services
- Is the IDL compiler essential?
  - No! Generic stubs and skeletons are possible, too.
  - Technique: Reflection and dynamic invocation

Reflection & Dynamic Invocation

- Reflection
  - to inspect the interface of an unknown component
  - for automatic / dynamic configuration of server sites
- Dynamic invocation
  - to call the components
- Problem
  - Language incompatibilities (solved)
  - Access to interfaces (open)
- Solution: IDL is already the standard
  - Standardize an IDL run time representation and access
  - Define an IDL for IDL representation and access

Example: Generic Server Skeleton Using Reflection

```
class GenericSkeleton extends Thread {
    static ExtendendHashtable objects = new ExtendedHashtable();
    ObjectOutputStream os;
    ObjectInputStream is;
    …
    public void run() { …
     Integer addr= (Integer) is.readObject(); //handler
     String  mn = (String)  is.readObject(); //method name
     Class[] pt = (Class[]) is.readObject(); //parameter types
     Object[] args= (Object[]) is.readObject(); //parameters
     Object  o = objects.getComponent( addr );
                  //object reference by reflective call
     Method m = o.getClass().getMethod( mn, pt );
                  //method object by reflection
     Object  res = m.invoke(o,args); //method call by reflection
     os.writeObject(res);
     os.flush(); s.close();
```


Services

- Predefined functionality standardized
  - Reusable
- Distinguish
  - Basic
    - Useful (only) with component services
    - Examples discussed: name and trader service
    - Further: multithreading, persistency, transaction, synchronization
  - General (*horizontal services*)
    - Useful (per se) in many domains
    - Examples: Printer and e-mail service
  - Domain specific (*vertical services*)
    - Result of domain analysis
    - Examples: Business objects (components)

Summary: What Classical Component Systems Provide

- Technical support: remote, language and platform transparency
  - Stub, Skeleton
    - One per component (technique: IDL compiler)
    - Generic (technique: reflection and dynamic invocation)
  - Adapters on client and server site
    - Generic (technique: Name services)
- Economic support: reusable services
  - Basic: name, trader, persistency, transaction, synchronization
  - General: print, e-mail, …
  - Domain specific: business objects, …
- More on these issues in the next lecture: CORBA