Objects

Service-Oriented Architecture

Jeremy Gibbons

Contents

1 Object-oriented middleware

- in a distributed system, components reside on different hosts
- hence operation requests must traverse a network
- ultimately, networks connect computers via the physical interchange (transmission and recognition) of electrical or physical signals
- . . . but that is nearly always too low a level to be convenient: higher level abstractions are to be preferred
- the ISO Open Systems Interconnection reference model is a layered architecture of such abstractions

1.1 ISO/OSI reference model

application layer: business logic

presentation layer: higher-level datatypes

session layer: connection protocol

transport layer: packets and streams

network layer: routing between nodes

datalink layer: network topology

physical layer: electrical and optical behaviour

1.2 Transport layer

- two main implementations on Unix systems
- *transmission control protocol* (TCP)
- *user datagram protocol* (UDP)
- different characteristics (connection-oriented vs connectionless), suitable for different kinds of application

1.3 TCP

- bidirectional byte streams between hosts
- connection-oriented
- buffering at both sides to decouple computation speeds
- reliable but slow
- really provides session-layer facilities too

1.4 UDP

- packet-oriented transmission
- packet contains destination address
- connectionless
- queueing at receiver only
- unreliable but fast
- session layer needs to provide reliability

1.5 The need for middleware

You could implement a distributed system over the transport layer:

- manual mapping of data values to byte stream
- manual resolution of data heterogeneity
- manual identification of components
- manual implementation of component activation
- manual synchronization of interaction
- no guarantees of type safety
- no guarantees of quality of service

. . . but you wouldn't want to.

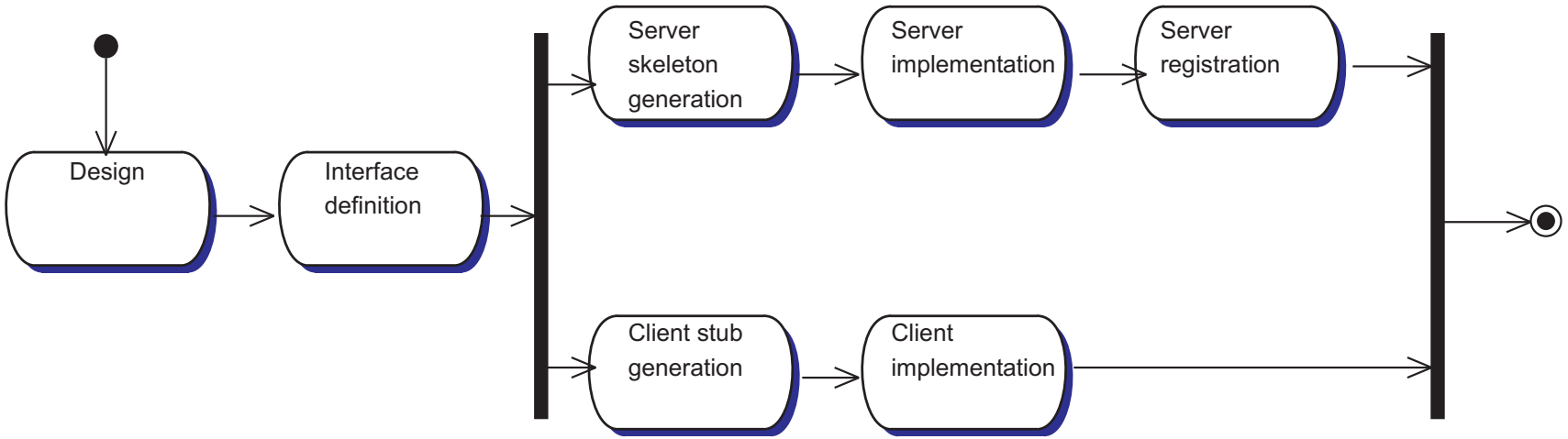
1.6 Middleware

- instead, these services are provided by *middleware*
- an implementation of the *session* and *presentation* layers of the OSI reference model
- bridge between application layer (‘business logic’) and transport layer (‘operating system’)
- makes distribution (largely) transparent
- different middleware philosophies, with different goals

1.7 Object-oriented middleware

- OO middleware evolved from remote procedure calls
- CORBA first; then DCOM, Java/RMI
- beyond RPCs, OO middleware supports object types as parameters, failure handling and inheritance
- *presentation layer* implementation must additionally define representation for object references, deal with exceptions, and marshall inherited attributes
- *session layer* implementation uses *object adapter* to map object references to objects (like `portmap`) and to activate inactive objects (like `inetd`)

1.8 Development steps



2 CORBA

‘Common Object Resource Broker Architecture’

- interface definition language

  language-independent specification of interfaces

- programming language independence

  multiple language mappings for IDL: legacy-friendly

- location transparency and automatic server activation

  call independent of physical location (which may vary dynamically); services started up on demand

- automatic stub and skeleton code generation for RPC

  network connections, marshalling, server forwarding

- reuse of CORBA services and facilities

  naming, transactions, security; compound documents; etc

2.1 Object Management Group

- CORBA standards produced by OMG
- world's largest computer industry consortium (now over 800 members)
- non-profit
- started 1989 with 3Com, American Airlines, Canon, Data General, HP, Philips Telecoms, Sun, Unisys
- doesn't develop technology or specs itself; provides structure whereby members specify and implement
- favours cooperation and compromise over choosing one member's solution

2.2 Object Management Architecture

- OMA is framework for all OMG technology
- two fundamental models on which CORBA and the other standard interfaces based
- *core object model*

  abstract definitions of concepts that allow distributed application development to be facilitated by an ORB; mostly of interest to ORB designers and implementers

- *reference model*

  places ORB at centre of groupings of objects with standardized interfaces that provide support for application object developers; relevant to CORBA programmers

2.3 Core Object Model

objects: models of entities or concepts; with *identity*, independent of properties or behaviour, and represented by an *object reference*

operations: *action* offered by object to outside world; has *signature* (name, parameters, result types); may cause *side-effects*, be *concurrent*, be *atomic*

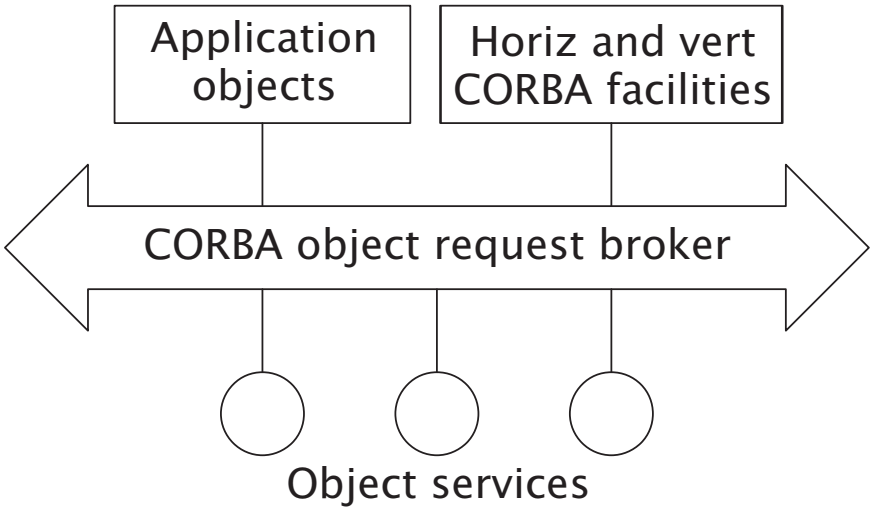
non-object types: numeric, strings, records, fixed-size arrays

interfaces: collection of operation signatures

substitutivity: being able to use one object in place of another without 'interaction error'; specifically, when interface of former includes that of latter

inheritance: explicitly behavioural substitutivity; hierarchy is rooted directed acyclic graph

2.4 Reference Model



object request broker: message bus for object invocations

object services: eg naming and trading, transactions, concurrency control, licensing, security

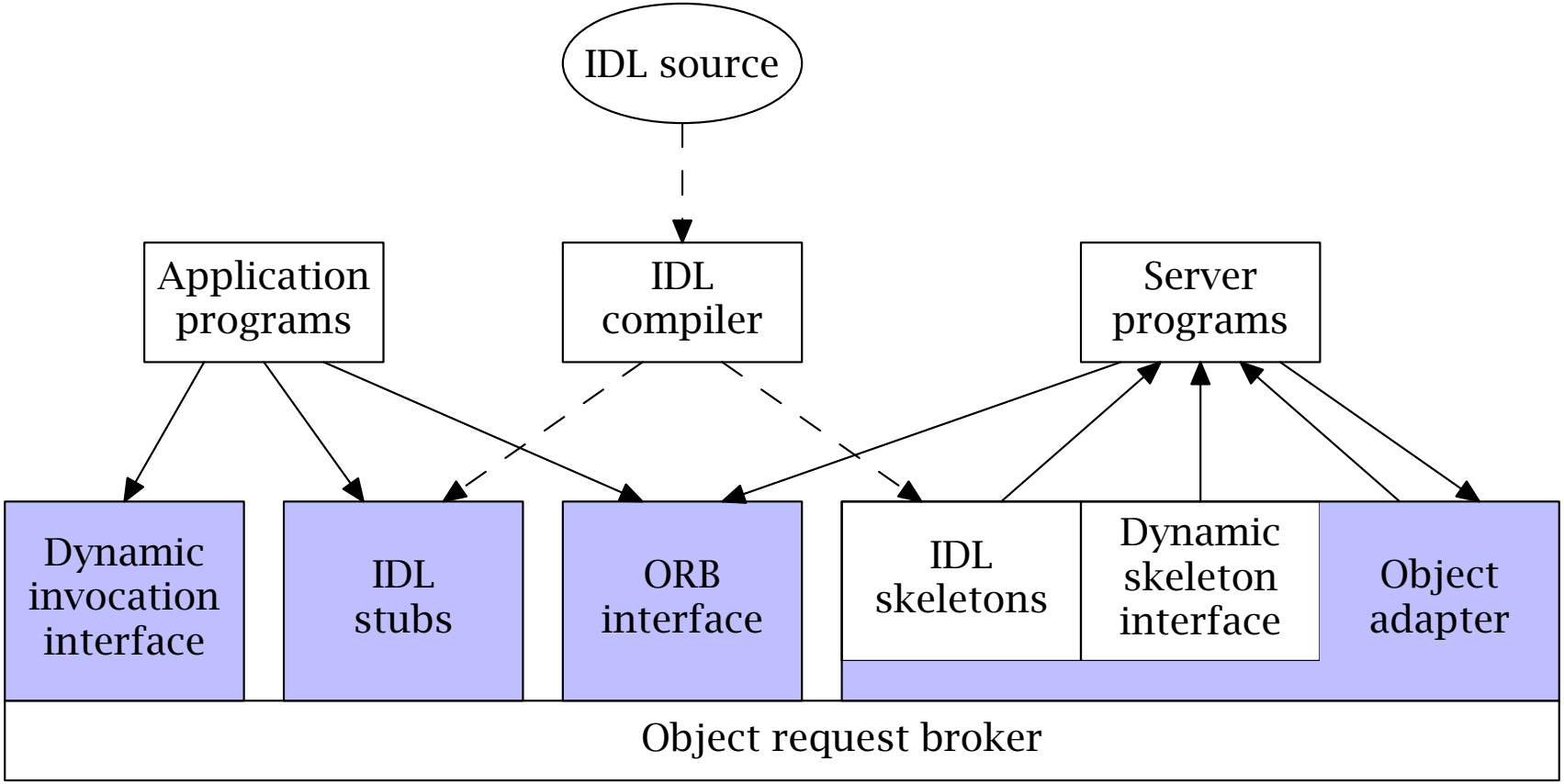
common facilities: horizontal (across application domains) and vertical (domain-specific)

application objects: competitive, but not standardized

2.5 OMG Interface Definition Language

- common language for specifying object interfaces
- types, modules, operations, interfaces, multiple inheritance
- *bindings* to common OO languages (Java, Smalltalk, C++, Ada95, even C and COBOL)
- IDL compiler
- deposit interface in ORB's *interface repository*
- register implementation with *implementation repository*

2.6 Structure of an ORB



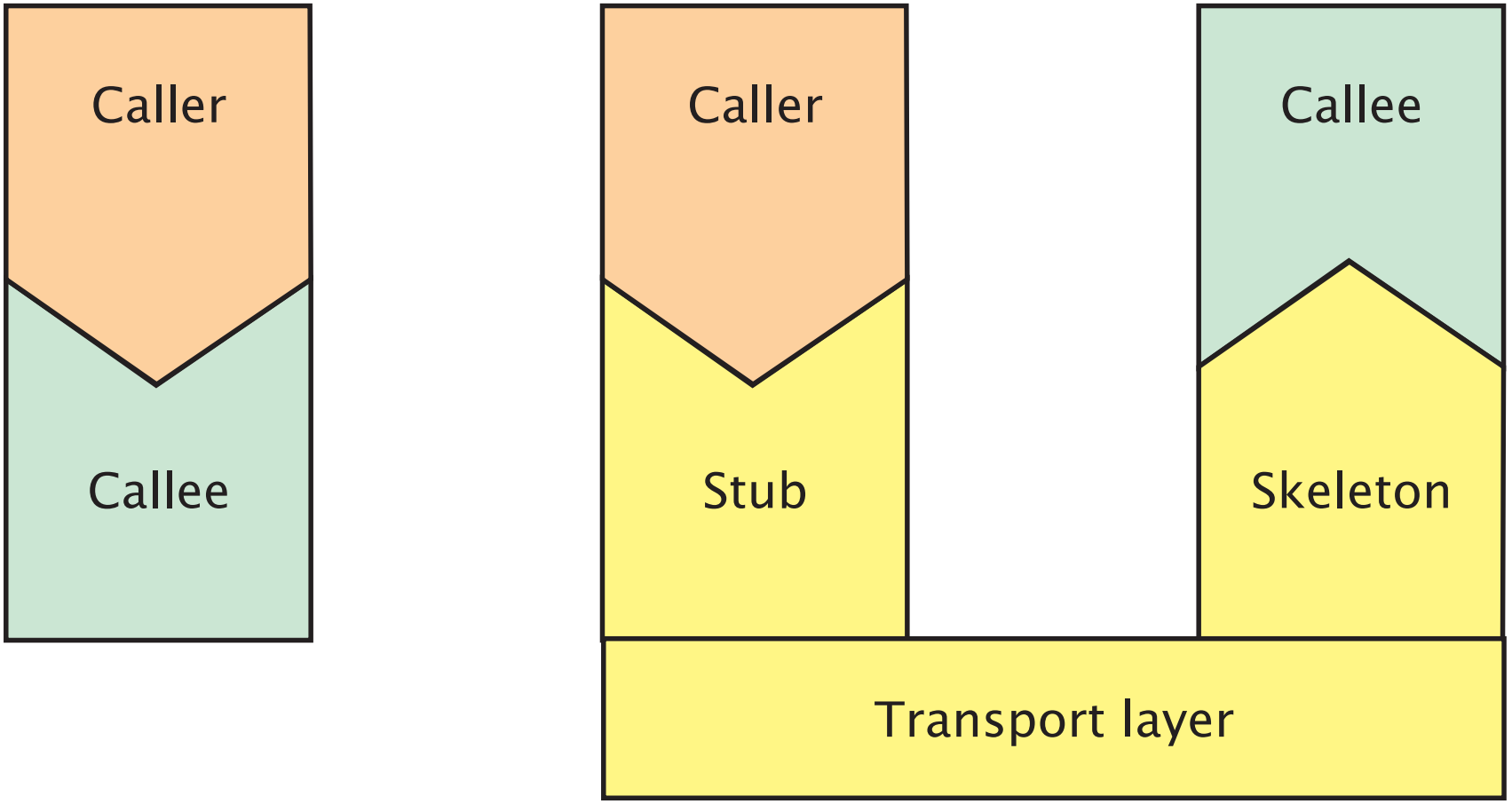
2.7 Client stubs

- collaborates with skeleton to implement presentation (and part of session) layer: heterogeneity, marshalling, streaming
- acts as proxy for client to invoke IDL-specified operation on object reference as if it were local method call
- client links in stub for IDL interface
- stub conveys invocation to remote target object
- generated from interface specification by IDL compiler for client's language

2.8 Implementation skeleton

- means to invoke correct method on correct implementation when request reaches server supporting some objects
- achieved by *implementation skeleton*
- automatically generated from interface specification by IDL compiler
- *unmarshalling* from wire-format to in-memory data structures (uses IDL language mapping), *invocation*, marshalling of *results*

2.9 Illustration of stub and skeleton



2.10 Generation Gap

- *Stub* and *Skeleton* typically generated automatically from IDL
- *Caller* needs to be linked with *Stub*, and *Callee* with *Skeleton*
- (always!) a bad idea to do this by editing generated code
- instead, generate (abstract) superclasses, and make modifications on subclasses
- an instance of Vlissides' GENERATION GAP pattern

2.11 Dynamic invocation interface

- DII allows invocation without having IDL compiler-generated stub
- client has object reference, constructs request dynamically
- provides operations to select target object for invocation, to name request, to add arguments
- also provides operations to invoke request and to retrieve results
- necessary information obtained from *interface repository*
- 'generic stub'

2.12 Dynamic skeleton interface

- allows ORB to invoke object implementation without compile-time knowledge of interface, ie without skeleton code
- idea is to invoke all implementations with same, general operation
- delaying dynamic selection till server-side
- object implementation cannot distinguish calls via compiler-generated skeleton from calls via DSI
- ‘generic skeleton’, eg single servant implementing many objects of different types (such as RDB entries)

2.13 Object adapters

- 'glue' used by object implementation to make itself available through ORB, and by ORB to manage run-time environment of implementations
- allows programming language entities to be registered as implementations for CORBA objects
- generates object references for CORBA objects
- activates server processes and objects if necessary
- early CORBA provided only a *Basic Object Adapter*, but this was underspecified and hence had incompatible implementations
- revisions since 2.2 provide also a *Portable Object Adapter*

2.14 Upper layers: services, facilities

- object services support CORBA-based programs independently of domain; fundamental infrastructure for distributed systems
- *common facilities* are component frameworks, horizontal (domain-independent) or vertical (domain-specific)

2.15 CORBAservices

change management: version tracking, compatibility

collections: sets, bags, queues, lists, trees, etc

concurrency: locks on resources

event notification: push or pull, over channels

externalization: serialization

licensing: licensing policies, usage monitoring and charging

lifecycle: creation, copying, mobility and deletion of objects

naming: association between names and unique IDs

persistence: save object instances on termination, restore later

properties: addition, retrieval, deletion of tags

queries: locate objects by attributes, using OQL and SQL

relationships: creation, deletion, navigation of relations

security: confidentiality, integrity, accountability, availability

time: clock synchronization

trading: locate services by description

transactions: atomicity, consistency, isolation, durability (ACID)

2.16 CORBAfacilities

- define component frameworks for integrating components
- *horizontal* facilities (domain-independent): focus on specific application models, unlike object services
- *user interfaces* (printing, email, scripting)
- *information management* (structured storage, universal data transfer, meta-data)
- *system management* (instrumentation, monitoring, logging)
- *task management* (workflow, rules, agents)
- eg *Distributed Document Component Facility* based on Apple's OpenDoc
- *vertical* common facilities, such as electronic finance, medical facilities, CAD

3 A simple CORBA example

- simplest possible distributed application
- ‘business logic’ says hello
- server makes business logic available
- client requests service

3.1 The interface

```
module helloworldexample {
  interface HelloWorld {
    string getMessage();
  };
};
```

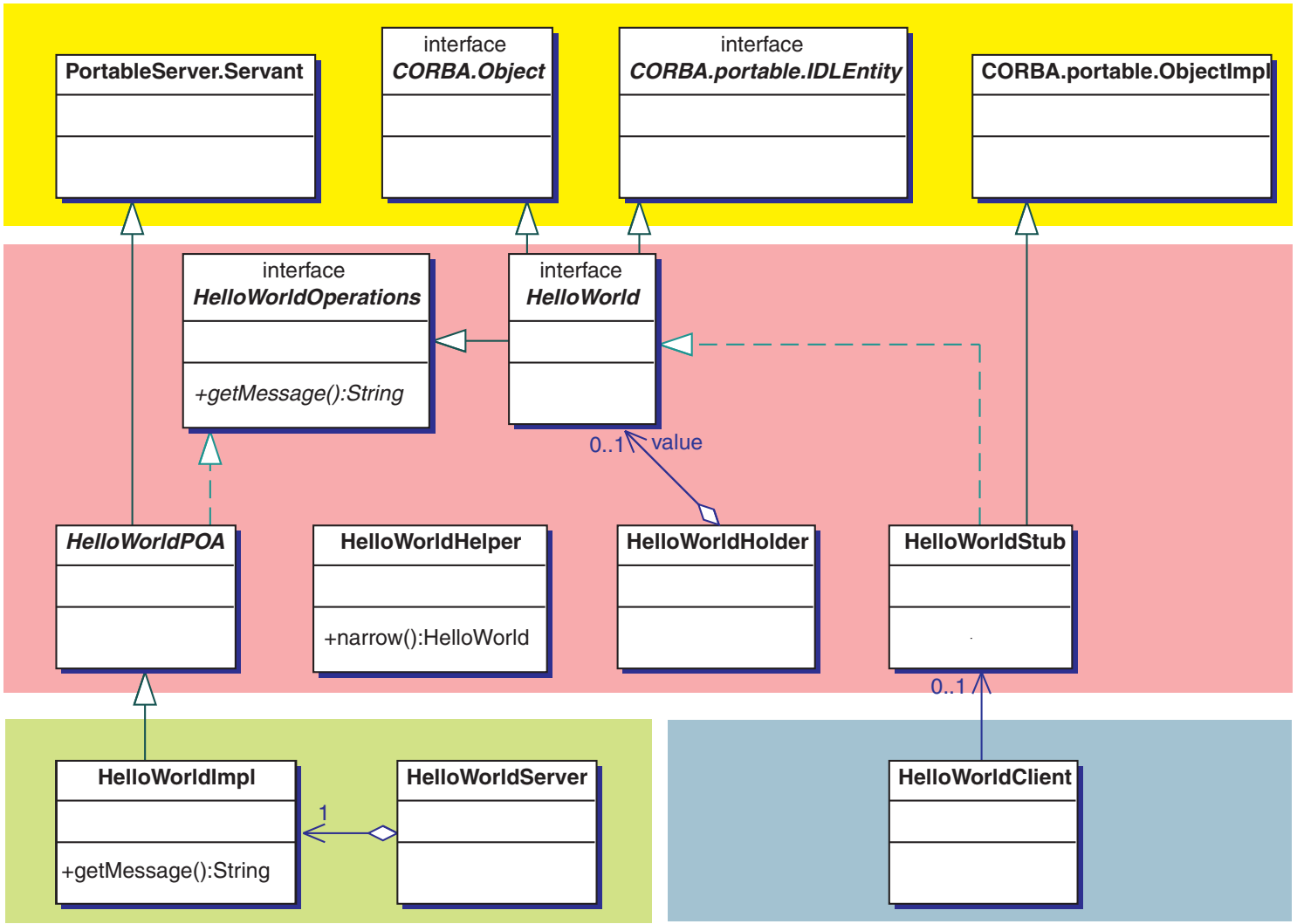
In this module, interface `HelloWorld` consists of a single method `getMessage`, which takes no parameters and returns a `string`.

3.2 Compiling the IDL

Running the IDL through the IDL-to-Java compiler generates a number of files (for each interface), in directory `helloworldexample` corresponding to module:

```
$ ls helloworldexample
HelloWorld.java
HelloWorldHelper.java
HelloWorldHolder.java
HelloWorldOperations.java
HelloWorldPOA.java
HelloWorldPOATie.java
_HelloWorldStub.java
```

3.3 Class diagram



3.4 Operations interface

Java translation of specified interface:

```
package helloworldexample;
public interface HelloWorldOperations {
  String getMessage();
}
```

3.5 Skeleton

Server-side object will extend this (abstract) skeleton:

```
package helloworldexample;
public abstract class HelloWorldPOA
  ... implements HelloWorldOperations {
  public ... _invoke(String opName, ...) {
    final String[] _ob_names = { "getMessage" };
    int _ob_index = ...; // find opName
    switch(_ob_index) {
    case 0: return getMessage(...);
    }
  }
  public abstract String getMessage();
}
```

3.6 Signature interface

Interface of what clients will actually receive:

```
package helloworldexample;
public interface HelloWorld extends
  HelloWorldOperations,
  org.omg.CORBA.Object,
  org.omg.CORBA.portable.IDLEntity {
}
```

3.7 Stub

Stub for client, implementing client interface:

```
package helloworldexample;
public class _HelloWorldStub ... implements HelloWorld {
  public String getMessage() {
    while(true) {
      try {
        ... _invoke("getMessage"...);
      }
      catch(...) {
        ...
      }
    }
  }
}
```

3.8 Helper

```
package helloworldexample;
final public class HelloWorldHelper {
  public static void
    insert(Any any, HelloWorld val) { ... }
  public static HelloWorld
    extract(Any any) { ... }

  public static TypeCode type() { ... }

  public static HelloWorld
    read(InputStream in) { ... }
  public static void
    write(OutputStream out, HelloWorld val) { ... }
```

```
    public static HelloWorld
      narrow(org.omg.CORBA.Object val) {
      try {
        return (HelloWorld)val;
      }
      catch(ClassCastException ex) {
      }
      ...
      return null;
    }
  }
```

3.9 Holder

To support call-by-name:

```
package helloworldexample;
final public class HelloWorldHolder ... {
  public HelloWorld value;

  public HelloWorldHolder() {}

  public HelloWorldHolder(HelloWorld initial) {
    value = initial;
  }

  ...
}
```

3.10 Fleshing out the skeleton

```
import helloworldexample.*;
public class HelloWorldImpl extends HelloWorldPOA {
  private String s;

  public HelloWorldImpl(String s) {
    this.s = s;
  }

  public String getMessage() {
    return s;
  }
}
```

3.11 The server

```
import helloworldexample.*;
public class HelloWorldServer {
  private static int run(...) { ... }

  public static void main(String args[]) {
    int status = 0;
    ORB orb = null;
    try {
      orb = ORB.init(args, null);
      status = run(orb, args);
    }
    catch(Exception e) { e.printStackTrace(); status = 1; }
    ...
  }
}
```

```
private static int run(ORB orb, String[] args)
  throws UserException {

  POA rootPOA = POAHelper.narrow(
    orb.resolve_initial_references("RootPOA"));
  POAManager manager = rootPOA.the_POAManager();

  HelloWorldImpl helloImpl = new HelloWorldImpl(args[0]);
  HelloWorld helloworld = helloImpl._this(orb); // alternatives!

  String ref = orb.object_to_string(helloworld);
  System.out.println(ref);

  manager.activate();
  orb.run();

  return 0;
}
```

3.12 The client

```
private static int run(ORB orb, String[] args)
  throws UserException {

  // Get "HelloWorld" object
  org.omg.CORBA.Object obj = orb.string_to_object(args[0]);
  if(obj == null) {
    System.err.println(
      "helloworldexample.Client: cannot unstringify IOR");
    return 1;
  }

  HelloWorld helloworld = HelloWorldHelper.narrow(obj);
  System.out.println(helloworld.getMessage());
  return 0;
}
```

3.13 Running the application

Start server:

```
java HelloWorldServer "Dib, dib, dib."
```

This prints out interoperable object reference:

```
IOR:000000000000002549444c3a68656c6c6f776f726c646578616d
706c652f48656c6c6f576f726c643a312e300000000000000000010000
0000000000780001020000000015736531372e636f6d6c61622e6f78
2e61632e756b000004060000000024abacab3139373732333133333400
5f526f6f74504f410000cafebabe3a3f5de6...
```

Then (in a different window, on a different machine) start client with this IOR:

```
java HelloWorldClient IOR:000...
```

This prints out the message

```
Dib, dib, dib.
```

3.14 What's in a name? iordump

```
byteorder: big endian
type_id: IDL:helloworldexample/HelloWorld:1.0
IIOP profile #1:
iiop_version: 1.2
host: se17.comlab.ox.ac.uk
port: 1054
object_key: (36)
171 172 171  49  57  56  57  53 "...19895" [...]
Native char codeset:
  "ISO 8859-1:1987; Latin Alphabet No. 1"
Char conversion codesets:
  "ISO 646:1991 IRV (International Reference Version)" [...]
Native wchar codeset:
  "ISO/IEC 10646-1:1993; UTF-16, UCS Transformation Format 16-bit"
Wchar conversion codesets:
  "ISO/IEC 10646-1:1993; UCS-2, Level 1"
```

4 Objects vs services

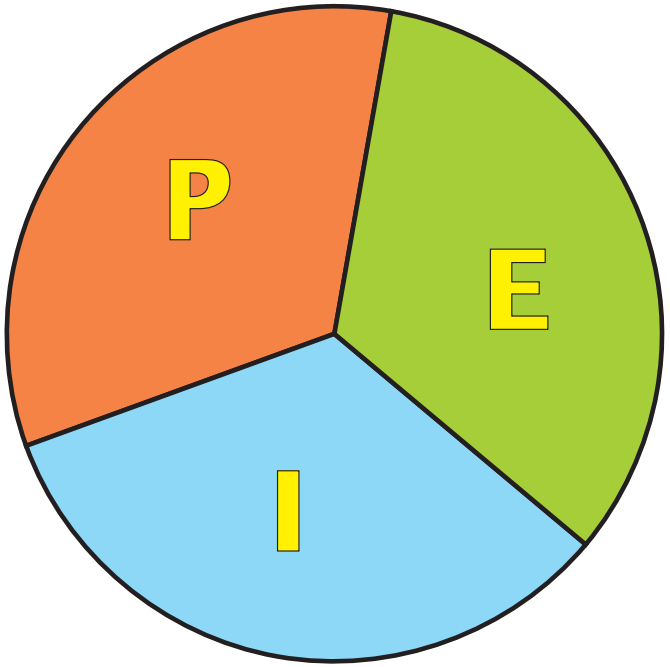
- CORBA promotes components, abstraction, message exchange, distribution...
- isn't this service-oriented too? what's different about SOA?

4.1 Object orientation

encapsulation: public *interfaces* and private *implementations*; hidden state

polymorphism: *multiple implementations* of interface, forming *subtypes* of interface type; instance of subtype is acceptable whenever instance of supertype expected

identity: each object has an *identity*, by which it is accessed; identities may be passed around, and persist even when object's internal state changes



4.2 Conversation

- many applications involve managing a *conversation*
- that is, they are stateful
- think of customer call centre
- works fine with distributed objects
  - obtain object reference
  - continuous dialogue with single object
  - object maintains state
- does not work so well with stateless services

4.3 Context

The essence of the problem is in maintaining shared context:

- what organization do you represent?
- where are we in this business process?
- what transactions have we done in the past?
- are there any resources that I promise to hold for you?
- are there any notifications I promise to deliver to you later?
- what permissions do you have?

4.4 Managing context

- implicitly: both parties keeping track

  fragile, redundant

- explicitly: passing to and fro

  inefficient, especially as relationship deepens

- externally: context as a resource

  For instance, on Expedia there is a ‘My Itinerary’ URI for each individual. Within that, every purchase you have recently made has its own URI. While you are purchasing a new ticket, each step in the process is represented by another URI. The client may keep copies of resources for its own protection, but the context is always mutually available as a series of linked documents at a URI.

  (Paul Prescod, *REST and the Real World*)

4.5 WS-Context

- grouping of related interactions into an *activity*
- nesting structures for related contexts
- *Context Service*
  - initiation, completion
  - expiry time too
- contexts can be passed by reference or by value
  - *Context Manager Service* for managing contexts-by-reference

4.6 WS-Resource Framework

- stateless access to stateful resource
- (not the same thing as conversational or session state)
- statefulness of resource made explicit in a standard way
- service makes (representation of) resource available
- declare, create, access, monitor, destroy

4.7 Identifiers

- uniform resource identifier (URI)
- services versus resources

- interoperable object reference (IOR)
- everything is an object: ‘services’ and ‘resources’
- references to either can be passed around
- callbacks, asynchronous messaging. . .

4.8 Documents

- SOA is *document-centric*
- focus is on the documents/representations
- services/transformations are transient
- dual to *object-oriented* approach
- focus is on the distributed objects
- transmitted data is transient
- maybe SOA is a misnomer?

Index

Contents

4.8 Documents

Service-Oriented Architecture

Monday	Tuesday	Wednesday	Thursday	Friday
Introduction Components	REST	Composition	Architecture	Engineering
coffee	coffee	coffee	coffee	coffee
Components	REST	Composition	Architecture	Conclusion
lunch	lunch	lunch	lunch	lunch
Web Services	Qualities	Objects	Semantic Web	
tea	tea	tea	tea	
Web Services	Qualities	Objects	Semantic Web	