Desarrollo de software basado en componentes y servicios

M.I. Capel

ETS Ingenierías Informática y
Telecomunicación
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada
Email: manuelcapel@ugr.es
http://lsi.ugr.es/mcapel/

October, 9th 2017

Máster Universitario en Ingeniería Informática

Fundamentals of component-based software development

Definition

Independently extensible system

Open systems:

- Concurrents
- Reactive
- Independently extensible
- Dynamic *input* and *output* of software-components in/out of a system

Open systems development and evolution problematics

Component

Definition

- A composition unit of software applications, which has a set of interfaces specified by contract and a set of requirements that only have *explicit context dependencies*.
- A software component can be developed independently, distributed and combined at run time.
- A component can be combined on time and space with other third party components.

Independent extension is the ability to introduce a new component without performing a global integrity check

Components vs. Objects

Caracteristics of a component

- Independent deployment unit
- Third party composition unit
- It cannot have an “externally visible” state
- There is no *implementation inheritance* of components

Caracteristics of an object

- Instantiation unit that only has a single indentity
- It has a state that can be externally visible
- It encapsulates state and behavior

Components vs. modules

- A module is a separated compilation unit
- Modules can implement TDA and have type checking
- Cannot be instantiated or defined (as *object references*)
- Modules that contain no classes can be also considered as components
- A module uses global static variables to externally show its state ($\neq$ component)
- Modules have an *excessive* external static dependence

- *Modularity* is prerequisite for the new *software component technology*
- A component needs independent extension and explicit control over dependencies, which modules cannot give

Components Based Programming

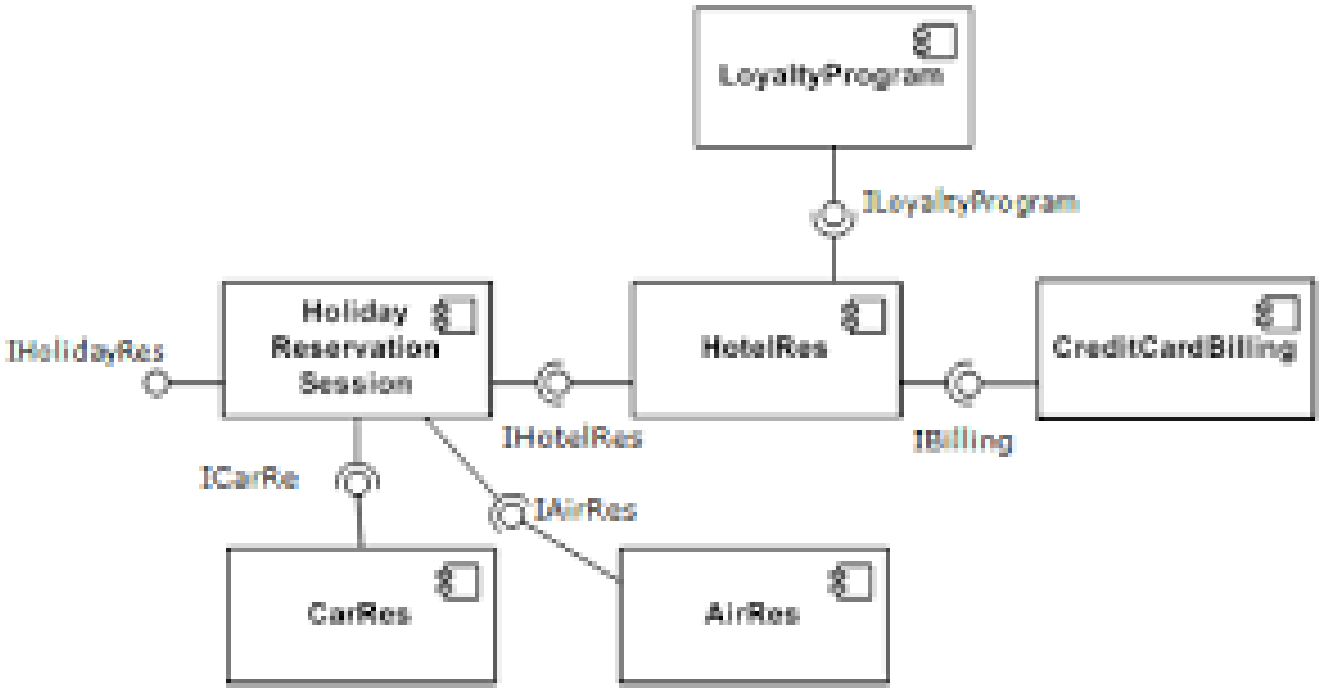
Definition

Distributed Programming (DP) and Component Based Programming (CBP) are based on a set of services that provide safe and efficient resource access to a component

Common types of services

- Remote communications
- Directory (name assignment, localization and access)
- Security
- Transactions
- Management (monitoring, handling and administration)

Component Based Programming - II



Concept

CBP is a natural extension OOP of use in *open systems* development, on which OOP shows limitations.

Component Based Programming - III

OOP limitations

- OOP does not differentiate between *computational* and *compositional* aspects in software aplications
- Drawbacks for reusing objects that have externally visible states
- Objects have *too low level* interfaces

Components

Definitions

- Composition unit with interfaces specified by contract and only explicit dependencies with its environment (Szyperski)
- Set of *atomic elements*={module +resources} simultaneously deployed
  (resources = “frozen collection of typed elements" that parameterise a component)
- Element that fulfills the 7 criteria of Meyer
- Binary unit that exports and imports functionality by using a standard interface mechanism (Stal)

Criteria of Meyer

The 7 criteria:

1. May be used by other software elements
2. May be used by clients without the intervention of the component's developer
3. Includes a specification of all dependencies
4. Includes a specification of the functionality it offers
5. Is usable on the sole basis of its specifications
6. Is composable with other components
7. Can be integrated into a system quickly and smoothly

Interfaces

- An interface must be interpreted as component contract: *what it does offer to its environment and what it receives in return*
- Interface= {atributes, public methods and events}
- *Signature* specification and event conditions
- We cannot *overspecify*: neither possible interface clients nor the interface way of use

Interoperability

To obtain interoperability between components:

- Standard specification of interfaces
- Discovery, localization y provisioning of services
- Component evolution handling

Interoperability II

Questions to be answered (in the future)

- One or several interfaces per object?
- Only interface implementation-based polimorfism
- Interface multiple inheritance?
- Naming and localization of services:
  - Unique identifiers (UUID, IID, CATID y CLSID)
  - Encrypted and marshalled resource localizators (CORBA IOR)
  - Directory services (`Registry` of JVM)or service associated *metainformation*?
  - Dynamic checking of interfaces by a *reflection* mechanism
  - Dynamic service creation

Components II

Common structure to all the definitions

- *Technical Part*: {contractual interfaces, compositionality, context independence}
- *Marketing Part*: “third party" role, deployment easyness
- *Invariant* setting (included in some definitions)

Specialized programming paradigm

Fundamental concepts of Component Oriented Programming

- Late composition
- Environments
- Events
- Reuse (white box, cristal, grey and black)
- Contracts
- Polimorfism: replaceability, parameterisation, binding
- Reliability
- Reflection

Concepts

Component-based model

1. Component interface shape
2. Interconnection mechanisms (between interfaces)
3. Plataforms assumed:
   -development and code execution environment
   -to isolate the most of drawbacks and specific aspects that comes with particular models of components

There is a strong relationship between models of components and frameworks

Standars

Evolution

1. Overcoming transportability problems of old OS-calls and primitive network systems
   Solution to procedure calls outbound of processes
   Only the Unix USB *sockets* were actually portable at the begining of 90's
2. First solution to the procedure-calls portability problem : RPCs
   Introduces new concepts: stubs, proxies, marshalling, etc.

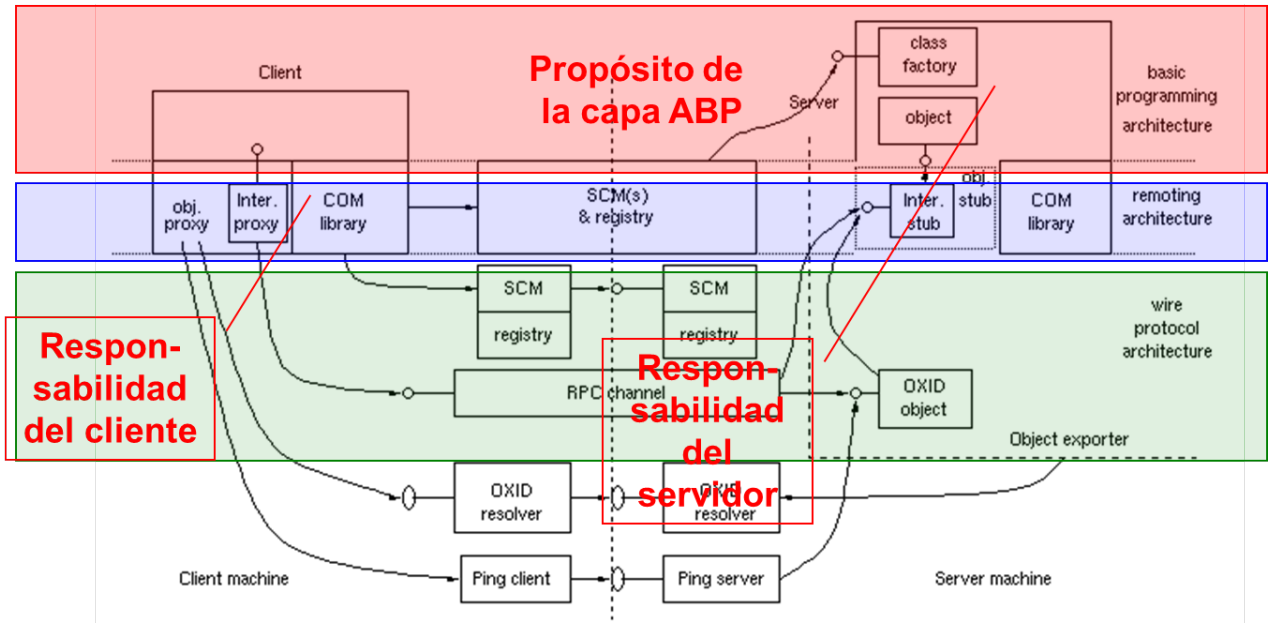
Standars II

Evolution

1. IDL and UUID proposed by DCE (OSF), etc.
2. Remote object calls problem
3. Dynamic libraries (DLLs) and the *"solution"* of Microsoft

Distributed Computing Environment (DCE) of OSF was the foremost service that implemented the RPC protocol for heterogeneous computation plataforms

DCOM: structure and function



3 fundamental layers:

- COM Server function
- COM objects and their interfaces
- Interaction between client and object

DCOM: fundamental concepts

Interface

"A named collection of abstract operations (or methods) representing a functionality"

Object Class

"A concrete and named implementation of one or more interfaces"

Object (Object Class Instance)

"A instantiation of an *Object Class*"

Standars III

Evolution

1. Calls to remote methods with late binding through CORBA ORB
2. Implementation of a *Virtual Machine* on the OS and the network: Java y .NET
3. Interoperability beyond the limits of the VM

CORBA

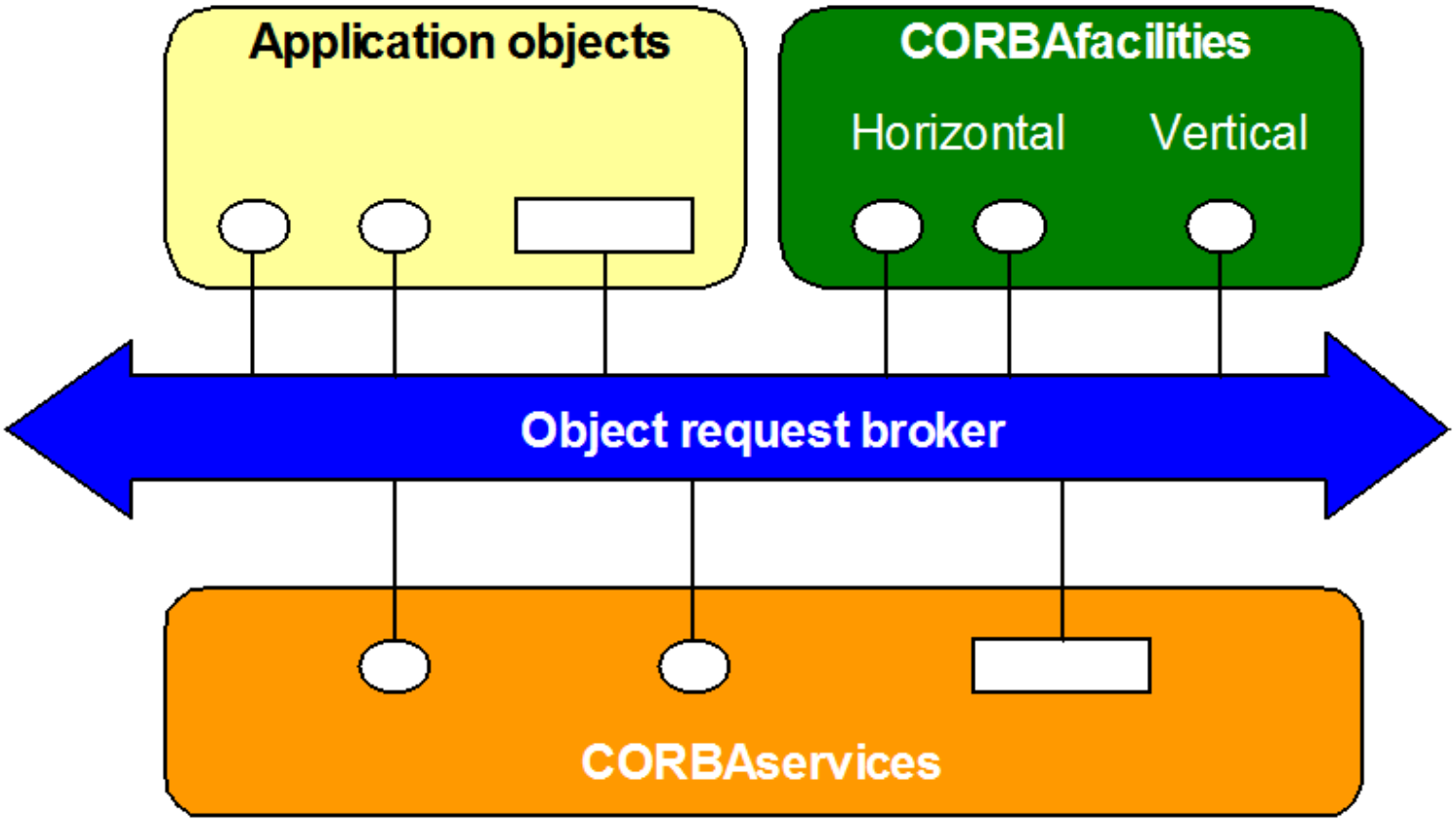


Figure: Software architecture for Heterogenous Distributed Systems

CORBA: the “object bus”

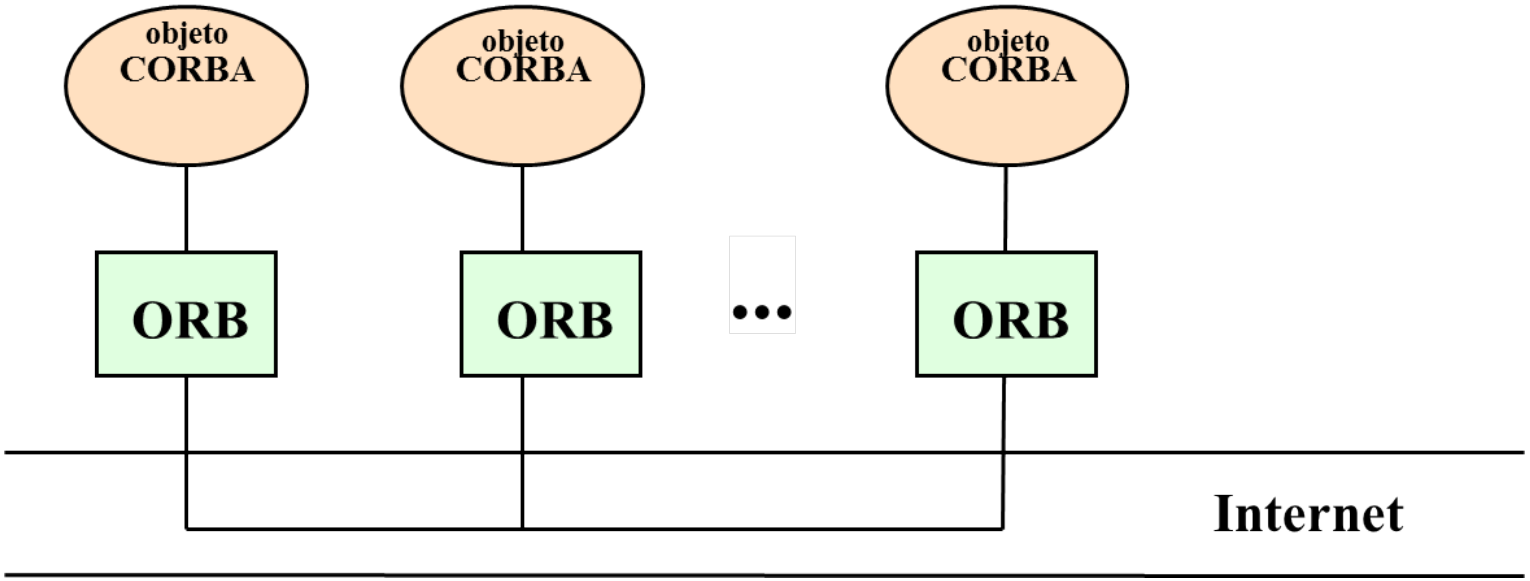


Figure: IIOP-receptive ORBs connected through Internet

CORBA application crossing platforms

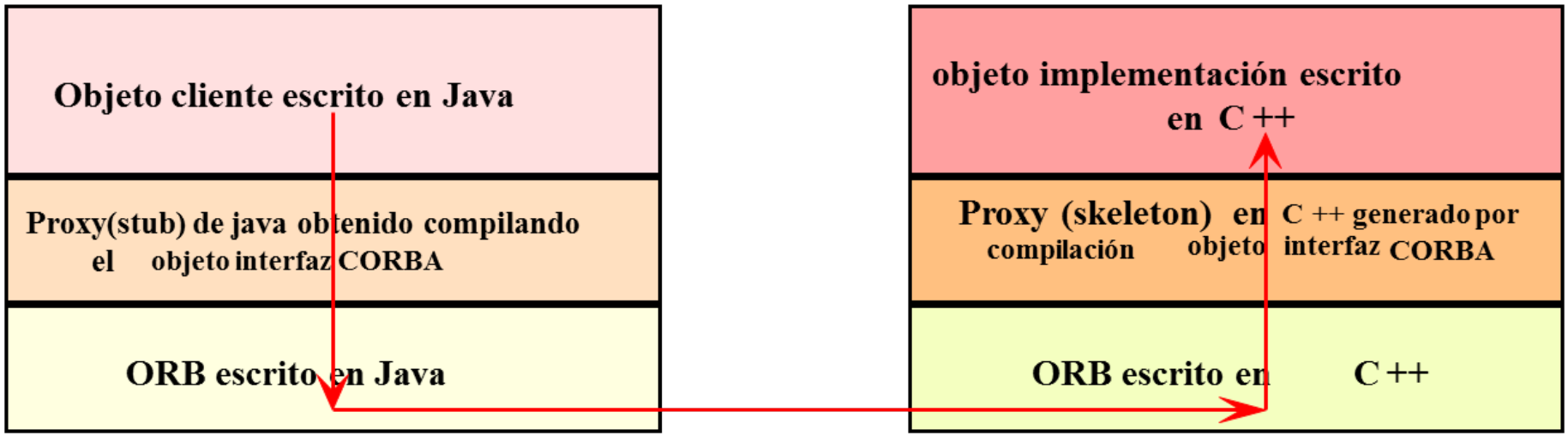


Figure: IDL and language interoperability

IDL: it is at a metalevel w.r.t. data, attributes, methods, interfaces, etc. of programming languages
IDL has a syntax similiar to Java or C++
IDL maps into programming languages, through an IDL compiler

Differences between DCOM and CORBA

- A DCOM client communicates with a COM object by obtaining a pointer to one of the object interfaces
- The client of a CORBA object obtains a reference to the remote object and uses the reference as a *handle* to make calls to the object's methods, as if the object was local to the client's address space
- CORBA also supports IDL-level multiple inheritance, which DCOM does not have

VM implementation

Virtual Machine

- A VM gives support for a uniform remote object calling mechanism, which is interoperable and portable, for each specific plataform
- Approach followed by Java Virtual Machine and the `.NET` environment
- JVM gives special support for interoperability beyond a local VM as well

Models of components II

Figure: Model and software components-based plataform

Examples of components platforms

Component Plataform

A development and code-execution environment for the isolation of the most part of *conceptual* and *technical* drawbacks that comes along with the development of component-based software applications, which follow a particular model of components

Model Plataform	COM ActiveX/OLE	JavaBeans EJB	CORBA Orbix

Containers

- The components exist and cooperate within a container(e.g.: ActiveX):
  - Shared interaction environment
  - Very convenient for wrapping visual objects
  - A reactive model of computation is adopted
  - A container is not modelled with objects easily
- Container–components relationships only through events
- A container can modify the graphical aspect of the components included in it
- A container can pass on object references to its included components

Framework

Reusable design of all (or a substantial part) of a system, which is made up by a set of abstract classes, and the way that these classes intercommunicate

Característics

- An application skeleton that must be adapted
- Set of entry points or “hooks", which make up the variable part of a software application
- Evolution and documentation problems that arise in software development with frameworks

RPCs-based solutions vs. Frameworks

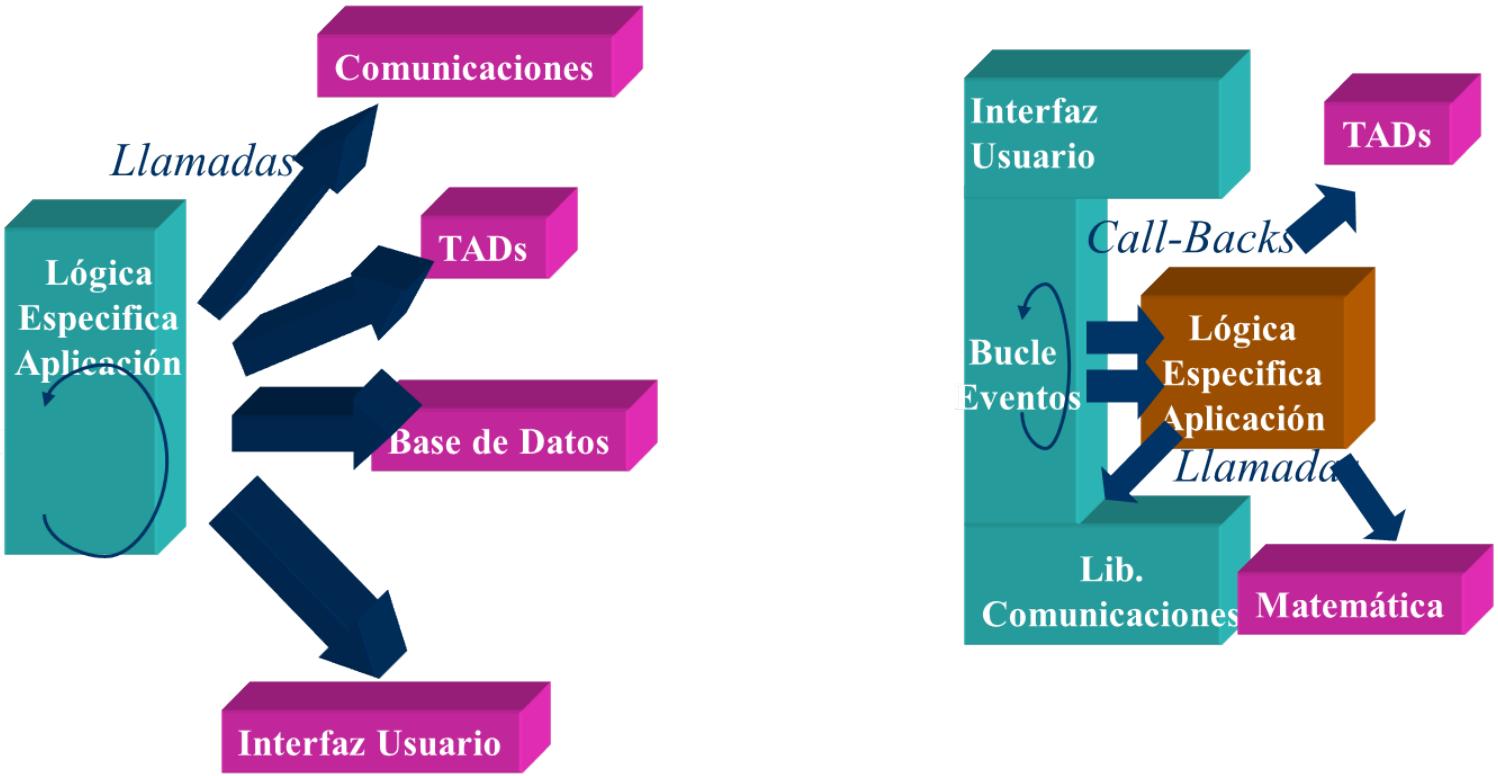


Figure: Differences between frameworks and library-based software development

Publish-Subscribe model for events

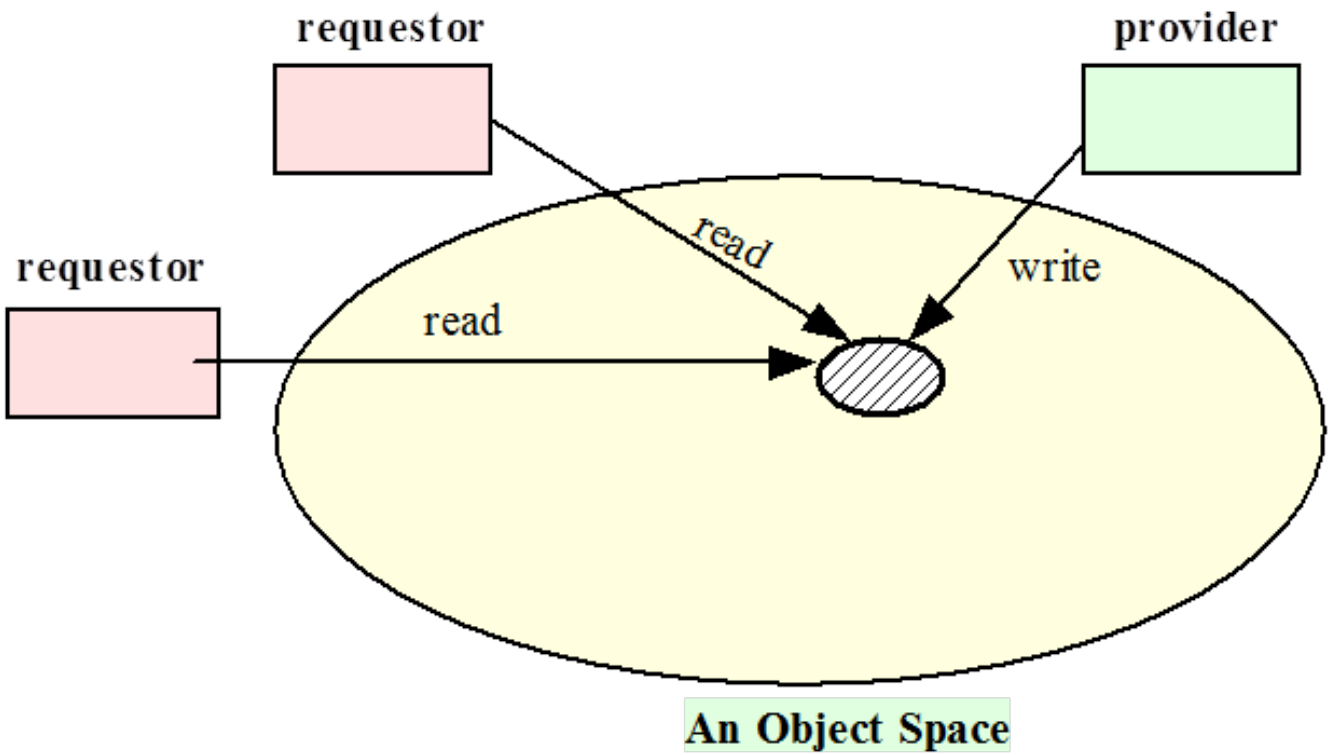


Figure: Distributed Computation Abstract Model

They can subscribe to the messages cast by this event
The middleware distributes the message to all the subscribers
powerfull abstraction for multicasting or group communication

Patterns

A design pattern gives an abstract solution to a common design problem A design pattern is defined through a set of relations and interactions between components

Useful characteristics

- For designing the architecture of frameworks
- The best possible documentation of a software architecture
- Domain dependent and independent catalogs of patterns

Software architectures

High abstraction-level representation of a system structure software application, which shows its integrating parts, the interactions among these, the design patterns that define the composition of the parts and the constraints at pattern application time

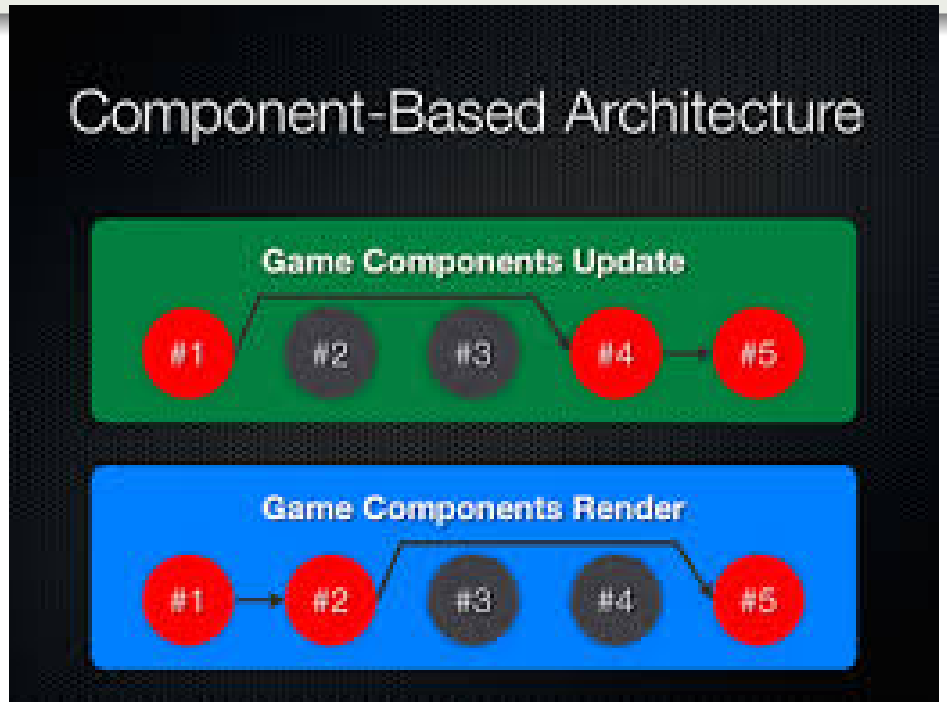


Figure: Components and architecture integration

Software architectures and components

Característics

- Collection of components and interactions
- To understand and to handle the structure of complex software applications
- To reuse that structure of any of its parts
- To plan the evolution of a software application

Component Oriented software development

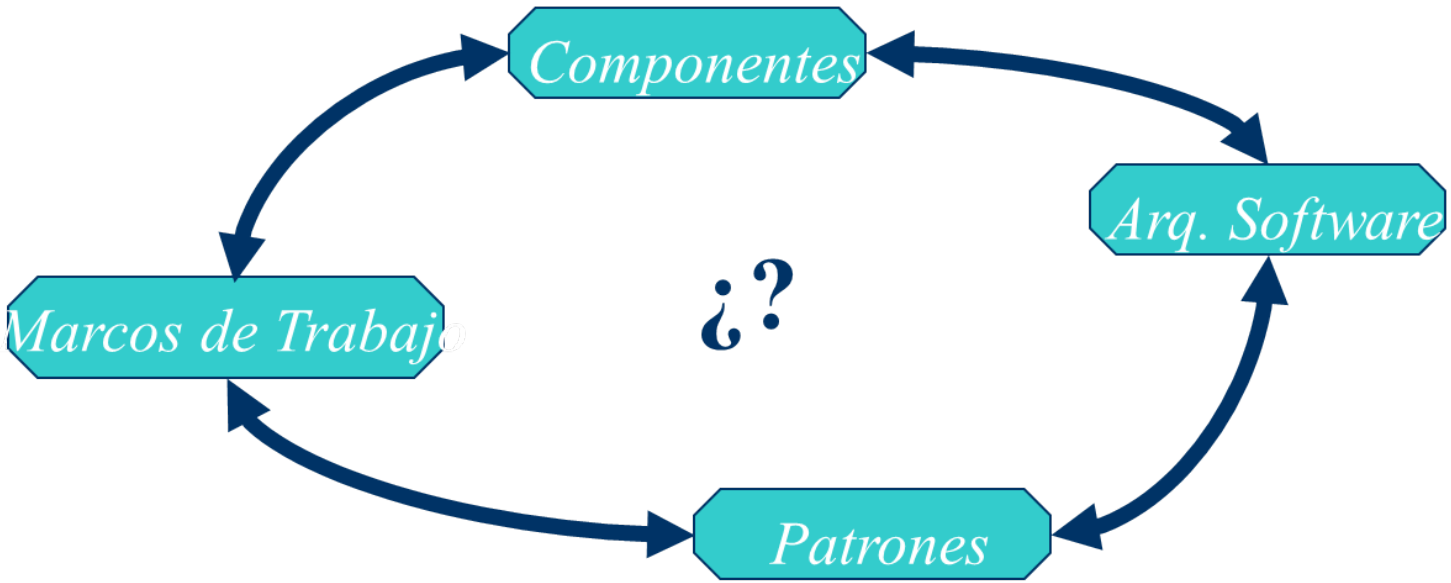


Figure: Simple ontology of component-based software

The missing piece is the “Middleware”

Models of components and UML

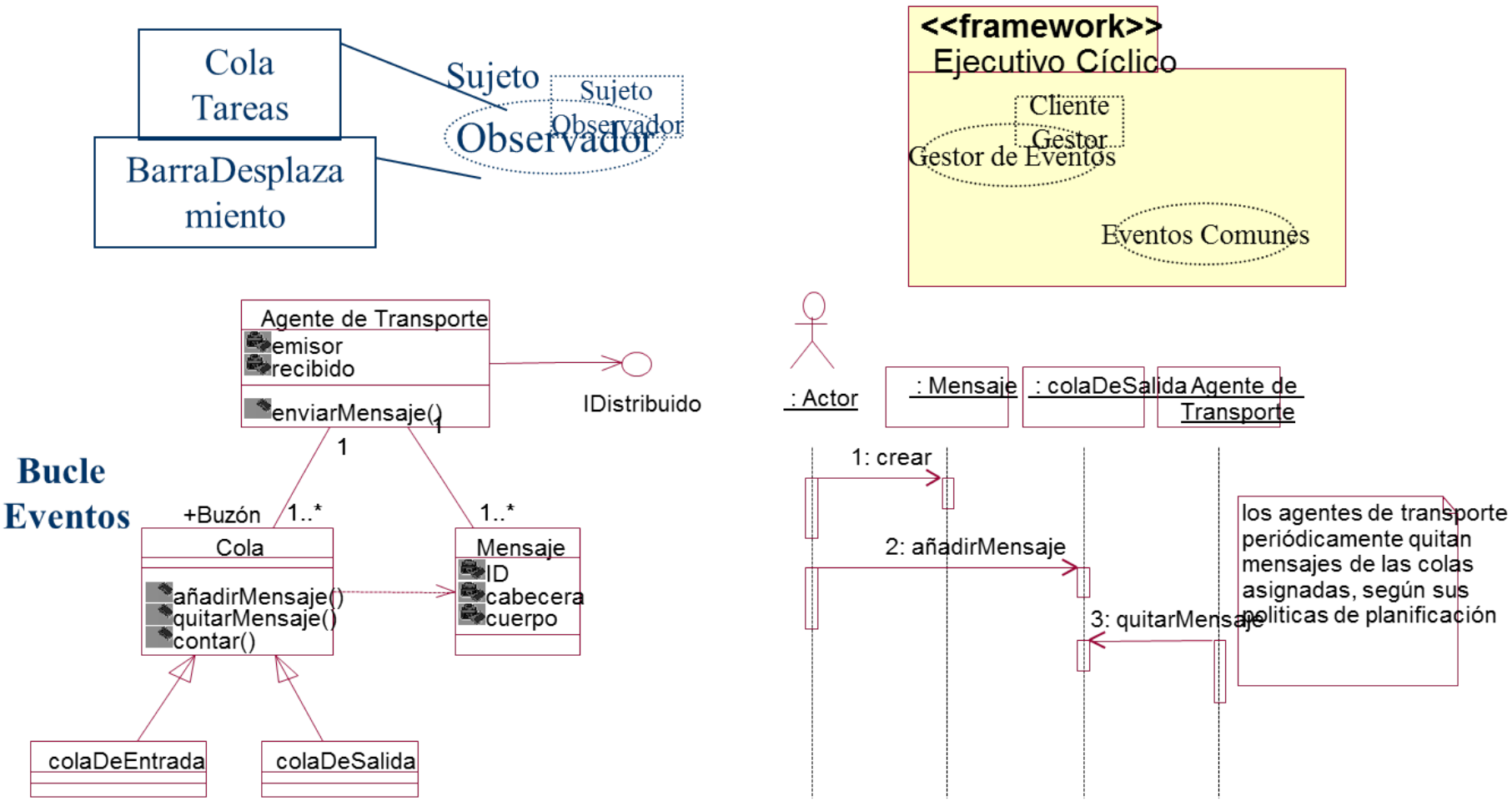


Figure: Middleware and UML mapping

Metainformation

This applies to all the information that a component has to disclose about itself and its properties

- Metainformation allows the discovery of all the functionality that components and containers give for handling by other components and containers
- Metainformation inspection can be static, dynamic and at run time
- Metainformation representation propitiates the use of *reflection*
- *Introspection* is an asset if a model of components supports *reflection*

IDEs

Definition

Visual application for building software applications from components

Elements of an IDE

- Canvas or container
- Editors for configuring and specializing components
- Viewers and browsers
- Directories of components
- Tools for component-based development (compilers, debuggers, unit proofs, etc.)
- Project management and control access
- Support for CSCW

Current IDEs examples

Name	**Builder**
Visual Studio	Microsoft
Visual Age	IBM
Visual Cafe	Symantec
Eclipse	ESF
NetBeans	Oracle

Complemented with configuration languages: JavaScript, VBScript, etc.

Bibliografía Fundamental


Eden, A., Hirshfeld, Y., and Kazman, R. (2006).
Abstraction classes in software design.
IEE Software, 153(4):163–182.


Exposito, D. and Saltarello, A. (2009).
Architecting Microsoft .NET solutions for the enterprise.
Microsoft Press, Redmond, Washington.


Szyperski, C. (1998).
Component Software. Beyond Object-Oriented Programming.
Addison–Wesley, **Básica**.


Verissimo, P. and Rodrigues, L. (2004).
Distributed Systems for System Architects.
Kluwer Academic.