Universidad Simón Bolívar

Departamento de Computación y T.I

Modelo de Objetos y Componentes Distribuidos

CORBA y Java Beans : casos de estudio

Prof. Yudith Cardinale

Septiembre-Diciembre 2011

Modelos de Programación

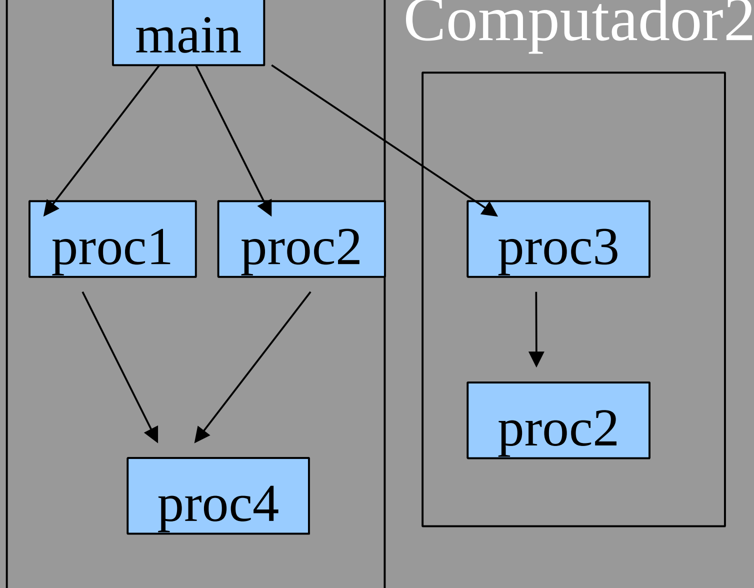
- Código Monolítico (Spaguetti)
- Distintos Archivos, Librerías
- Objetos
  - Ventajas: código más modular, reusabilidad, mantenibilidad, división del trabajo.

Modelos de Programación para Aplicaciones Distribuidas

- Llamadas a procedimientos remotos (RPC, no se usa en modelos de objetos distribuidos)
- Invocación a un método remoto (RMI)
- Modelo de programación basado en eventos (ejemplo Java Beans)

Modelos de Programación para Aplicaciones Distribuidas: RPC

Computador1



Comp. local

Comp. remoto

Funciones del servidor

Cliente

func()

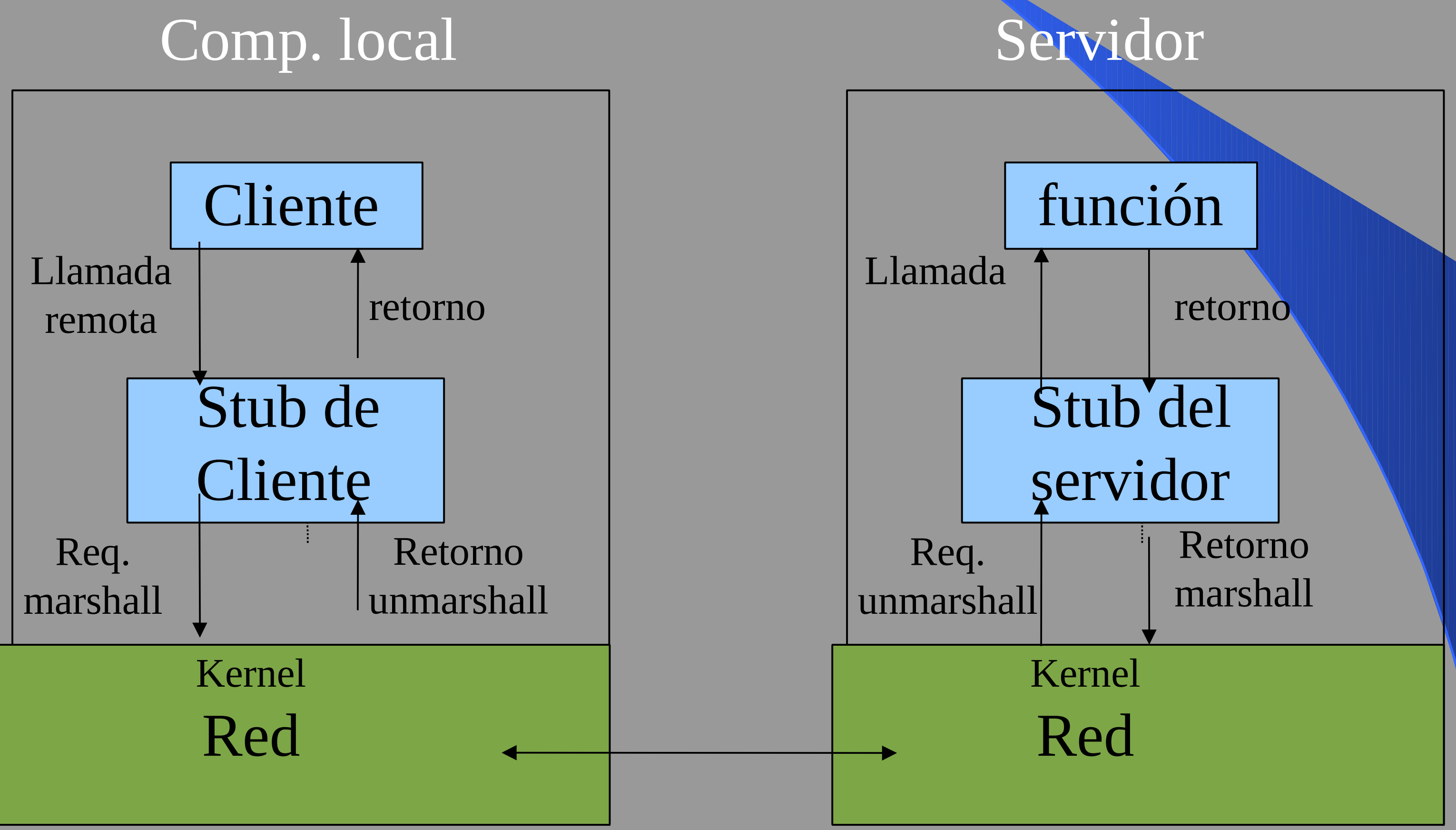
func() {
...
}

Hilo de ejecución

Hilo bloquead

RPC

Modelos de Programación para Aplicaciones Distribuidas: RPC



Modelos de Programación para Aplicaciones Distribuidas: objetos

Comp. local

Cliente

Result = obj.method1(val)

Servidor

```
Def obj {
  int method1(int val)
     {…}
  int method2(string s)
     {…}
  int method3(int v)
     {...}
}
```

Interfaces

Interfaces en los sistemas distribuidos

- Interfaces de servicio: especificación de servicios que ofrece un servidor (atributos de cada uno)
- Interfaces remotas: especificación de métodos remotos (los objetos pueden ser argumentos)
- Los lenguajes de definición de interfaces (IDL) están diseñados para permitir que los objetos implementados en lenguajes diferentes se invoquen unos a otros. Ejem: CORBA IDL, Sun XDR.

Modelo de Objetos

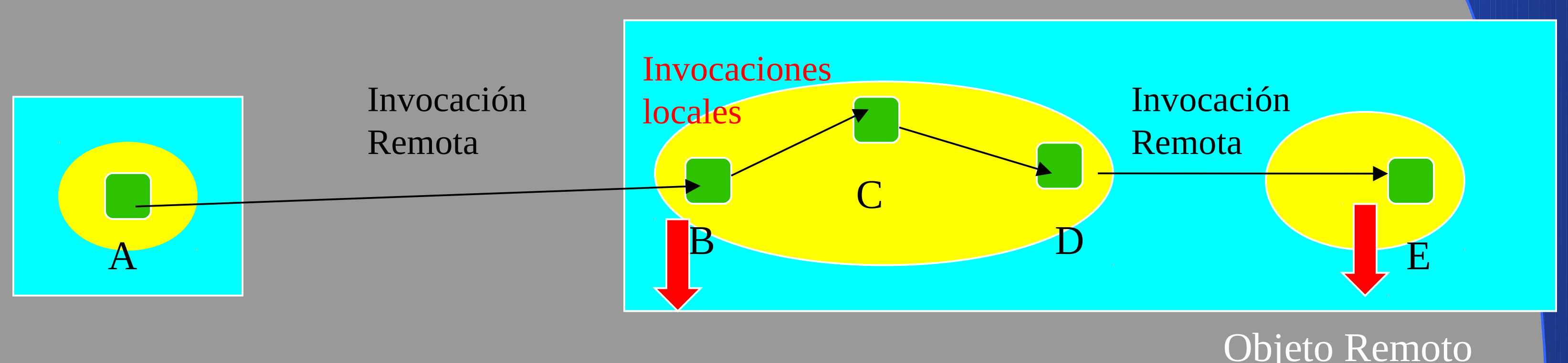
- Un programa OO consta de un conjunto de objetos que interactúan entre ellos.
- Cada objeto se compone de un conjunto de datos y un conjunto de métodos.
- Un objeto se comunica con otro objeto invocando sus métodos, generalmente pasándole argumentos y recibiendo resultados
- Se puede acceder a los objetos mediante referencias a objetos.

Modelo de Objetos

- La interfaz define las firmas o *signatures* de los métodos (tipos de sus argumentos, valores devueltos y excepciones) sin especificar su implementación.
- Se manejan excepciones
- Se hace necesario proporcionar mecanismos de liberación del espacio ocupado por aquellos objetos que ya no lo necesitan (compactación automática de la memoria)
- El estado de un objeto distribuido reside en una sola máquina, las interfaces están disponibles en varias máquinas

Modelo de Objetos Distribuidos

- Cada proceso contiene un conjunto de objetos, algunos de los cuales pueden recibir tanto invocaciones locales como remotas. Otros objetos sólo pueden recibir invocaciones locales.



Modelo de Objetos Distribuidos

- La invocación se lleva a cabo ejecutando el método del objeto en el servidor y el resultado se devuelve al cliente en otro mensaje.
- El estado del objeto reside en una sola máquina, sólo las interfaces están disponibles en varias máquinas

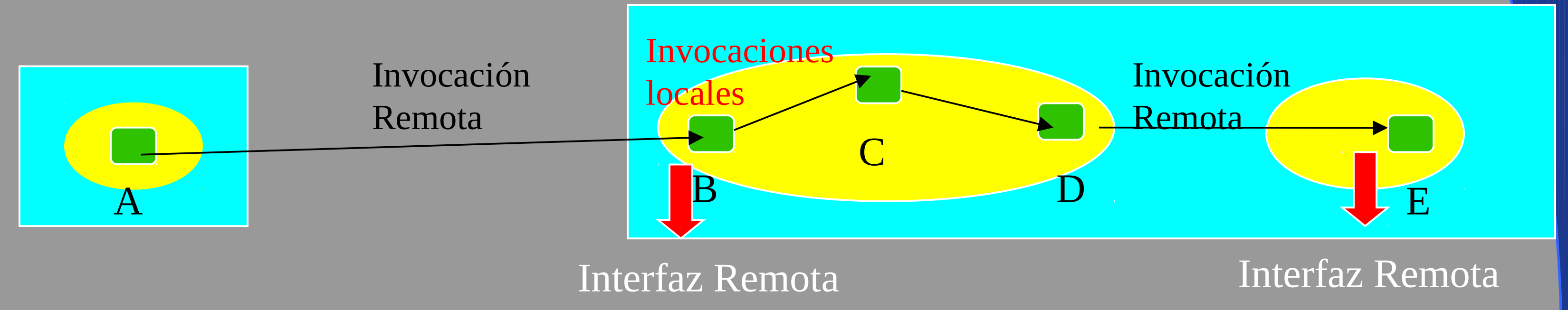
Modelo de Objetos Distribuidos

Existen dos conceptos fundamentales:

- Referencia de objeto remoto:
  - Otros objetos pueden invocar los métodos de un objeto remoto si tienen acceso a su referencia de objeto remoto.
  - Las referencias a objetos remotos se pueden pasar como argumentos y resultado de las invocaciones de métodos remotos.

Modelo de Objetos Distribuidos

- Interfaz remota:
  - Cada objeto tiene una interfaz remota que especifica cuáles de sus métodos pueden invocarse remotamente.
  - El sistema CORBA proporciona un IDL que permite definir interfaces remotas. Las clases de los objetos remotos y los programas de los clientes pueden implementarse en cualquier lenguaje. Los clientes CORBA no tienen que estar en el mismo lenguaje que el objeto remoto.



Modelo de Objetos Distribuidos

- La compactación automática de memoria distribuida, se logra usualmente mediante la cooperación entre el compactador de memoria local y un módulo adicional que realiza compactación de memoria distribuida, basado en un contador de referencias.
- Una invocación a un método remoto debe ser capaz de lanzar excepciones. CORBA IDL proporciona una notación para las excepciones.

Objetos en tiempo de compilación vs. objetos en tiempo de ejecución

- Objetos en tiempo de compilación:
  - Son objetos soportados por el lenguaje de programación (C++, Java, etc). En este caso el objeto se define como una instancia de una clase. Una clase es una descripción de un tipo abstracto de datos.
  - Un objeto se define mediante su clase y las interfaces que implementa. Las interfaces pueden compilarse en resguardos del lado del cliente y del servidor, lo cual permite la invocación remota de objetos.
  - Desventaja: dependencia de un lenguaje de programación.
  -

Objetos en tiempo de compilación vs. objetos en tiempo de ejecución

- Objetos en tiempo de ejecución:
  - Otra forma de construir objetos es hacerlo a tiempo de ejecución. Se puede implementar de cualquier forma.
  - Puede ser una librería de funciones en C que acceden a un archivo de datos, pero cómo hago que parezcan métodos de un objeto???
  - Uso un adaptador de objetos, es como una envoltura sobre la implementación para darle forma de objeto. El adaptador “se comunica” con la librería de C y abre un archivo de datos que representa el estado actual del objeto.

Objetos en tiempo de compilación vs. objetos en tiempo de ejecución

- Objetos en tiempo de ejecución:
  - Este método se sigue en muchos sistemas distribuidos basados en Objetos (Corba). Es independiente del lenguaje de programación en el que están escritas las aplicaciones distribuidas. Los objetos pueden escribirse en varios lenguajes.
  - Los objetos deben registrarse con un adaptador quién posteriormente hace que la interfaz esté disponible para invocaciones remotas.

Objetos persistentes vs objetos transitorios

- Objetos persistentes: continúan existiendo aun cuando ya no esté contenido en el espacio de direcciones de cualquier proceso servidor. El servidor puede almacenar el objeto en memoria secundaria y después terminar.
- Objetos Transistorios::
  - Existe sólo en tanto el servidor que lo está alojando exista.
  - Se pueden crear a la primera solicitud de invocación y destruirlo en cuanto ya ningún cliente esté ligado a tal objeto.
  - Crear todos los objetos transitorios en el momento en el que el servidor se inicializa (ningún cliente puede utilizar el objeto

Servidores de Objetos

- Un servidor de objetos es un servidor diseñado para soportar objetos distribuidos.
- La diferencia más importante entre un servidor general y un servidor de objetos, es que un servidor de objetos no proporciona por sí mismo un servicio específico.
- Los servicios son implementados por los objetos que existen en el servidor. Al cambiar los objetos cambian los servicios.

Servidores de Objetos

- Un servidor puede tener un solo hilo de control para atender todas las solicitudes.

- Tener varios hilos de control, uno por cada objeto que maneja. Si llega una solicitud al método de un determinado objeto se pasa al hilo correspondiente, si está ocupado, se encola la solicitud.

- Los objetos quedan protegidos contra acceso concurrente, las invocaciones se serializan a través del hilo correspondiente.

Servidores de Objetos

- También es posible utilizar un hilo por cada solicitud de invocación. En este caso los objetos deben estar protegidos para el acceso concurrente.
- Crear hilos por demanda o tener un conjunto fijo de hilos.

Corba: un estándar para construir objetos distribuidos.

Object Management Group (OMG)

- Es un consorcio internacional que promueve el desarrollo de software orientado por objetos (http://www.omg.org)

- El objetivo del OMG es proveer un marco de arquitectura común para permitir la interacción de objetos en plataformas heterogéneas y distribuidas.

Object Management Group (OMG)

- Fue fundado en 1989.
- Inicialmente estuvo conformado por 8 compañías: 3Com Corpotation, American Airlines, Canon Inc., Data General, Hewlett-Packard, Philips Telecommunications N.V., Sun Microsystems y Unisys Corporation.
- Actualmente hay más de 1000 miembros

Object Management Group (OMG)

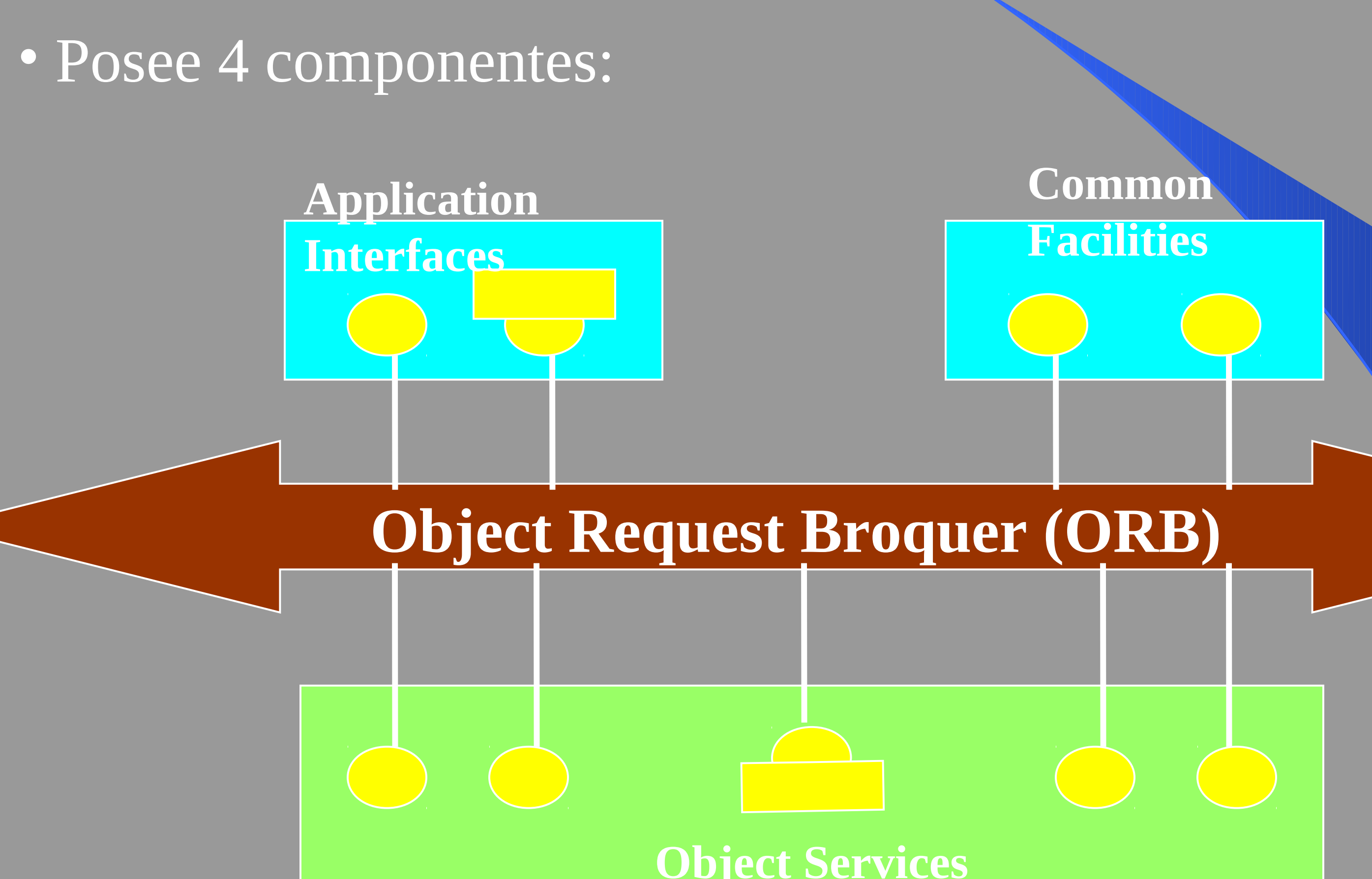
- El OMG no realiza trabajos de desarrollo e implementación, más bien se basa en la tecnología existente ofrecida por sus miembros.
- Propone especificaciones para el desarrollo de computación distribuida basada en objetos: OMA (Object Management Architecture).
- Los miembros pueden proponer especificaciones, luego de un proceso de revisión y votación podrán incorporarse al OMA.

Object Management Architecture (OMA)

- OMA es una arquitectura de referencia sobre la cual se pueden construir aplicaciones.
- Define, a un nivel alto de abstracción las “facilidades” necesarias para el desarrollo de aplicaciones distribuidas orientadas por objetos.

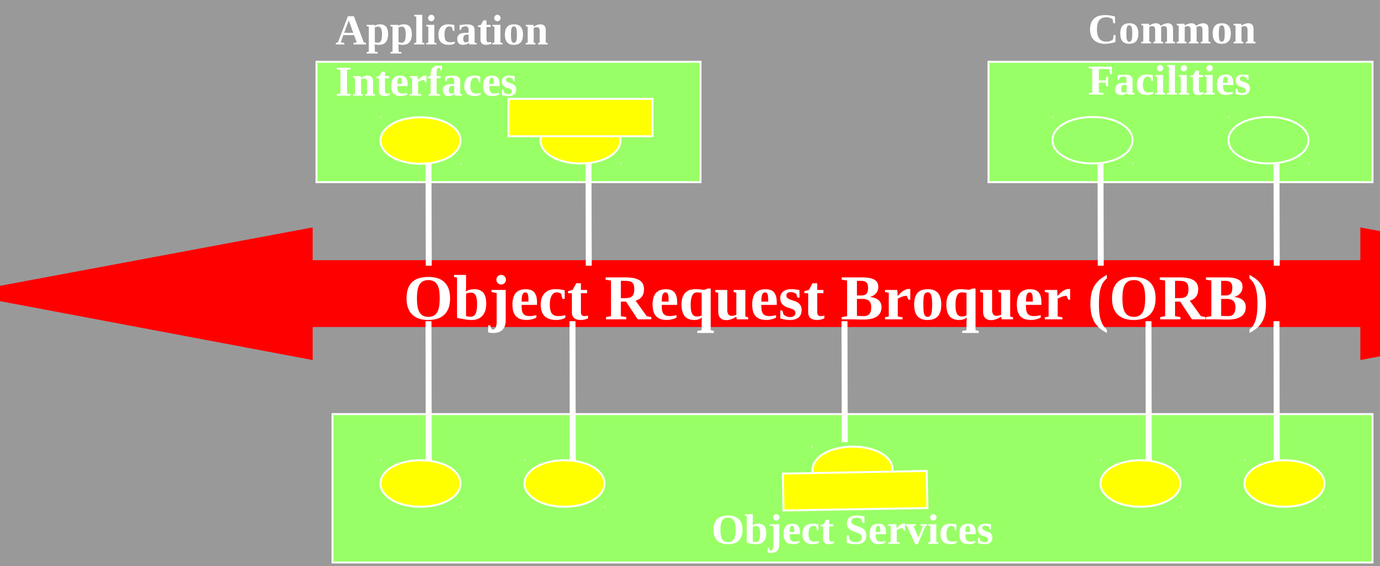
Object Management Architecture (OMA)

- Posee 4 componentes:



Object Management Architecture (OMA)

- **ORB** (intermediario de petición de objetos) es el bus de comunicación entre objetos. Permite o facilita la comunicación entre objetos. La tecnología adoptada para los ORBs es lo que se conoce como CORBA.

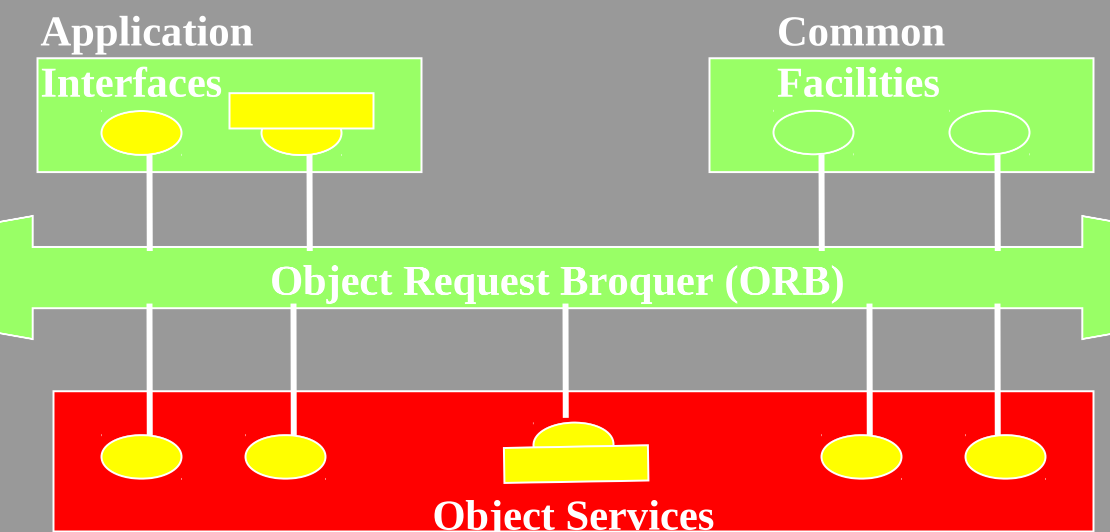


Object Management Architecture (OMA)

- CORBA (***Common Object Request Broker Architecture***) es un estándar para construir objetos distribuidos.
- Es un diseño de middleware que permite que los programas de aplicación se comuniquen unos con otros con independencia del lenguaje de programación, sus plataformas de hw y sw, las redes sobre las que se comunican y sus implementadores.

Object Management Architecture (OMA)

- ***Object Services***: definen un conjunto de objetos que implementan servicios fundamentales de bajo nivel como servicio de nombres, seguridad, persistencia, ejecución concurrente y transacciones, etc. Permiten a los desarrolladores construir aplicaciones sin tener que reinventar la rueda.



Object Management Architecture (OMA)

- ***Common Facilities (CF):*** son servicios de más alto nivel orientados a las aplicaciones (accounting, intercambio de información entre aplicaciones, correo electrónico o facilidades para imprimir).
- ***Application Interfaces:*** interfaces desarrolladas para aplicaciones en particular. Estos objetos hacen uso del resto de los componentes. OMG probablemente no llegue a desarrollar estándares de este tipo.

Enfoque CORBA

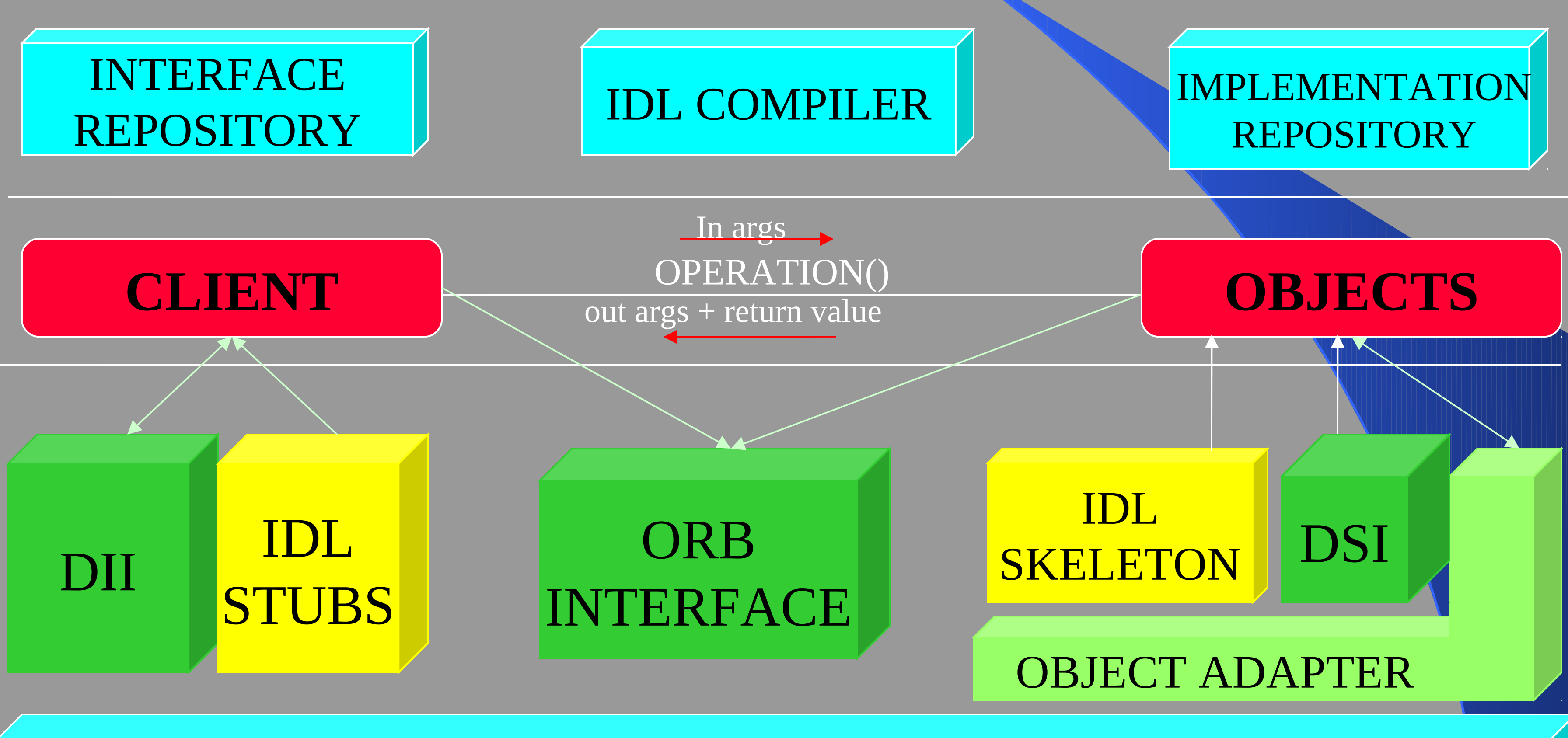
- Especifica protocolos básicos (GIOP/IIOP) usados para tener acceso remoto a los objetos.
- CORBA 1.1 (1991) propone una implementación específica de ORB (IDL)
- CORBA 2.0 (1994) propone interoperabilidad entre ORBs proponiendo estándares para permitir la comunicación entre implementaciones realizadas por desarrolladores diferentes. Este estándar se denomina GIOP (General Inter-ORB Protocol).
- CORBA 3.0 (1997) incrementa interoperabilidad y funcionalidad (POA).
- CORBA 3.0.2 (2002) última versión.

Enfoque CORBA

- Los clientes no son necesariamente objetos; un cliente podrá ser cualquier programa que envía mensajes de petición a objetos remotos y reciba respuestas.
- El término objeto CORBA se refiere a objetos remotos.
- Un objeto CORBA implementa una interfaz IDL, tiene asociado una referencia de objeto remoto y es capaz de responder a las invocaciones de los métodos en su interfaz IDL.
- Se puede implementar un objeto CORBA en un lenguaje que no sea OO.

Arquitectura CORBA

Componentes en el modelo de referencia



Arquitectura CORBA: IDL

IDL: ***Interface Definition Language***

La interfaz especifica un nombre y un conjunto de métodos que podrán utilizar los clientes.

- Es un lenguaje declarativo
- Define tipos de objetos especificando sus interfaces estáticas
- Sintaxis derivada de C++ con palabras adicionales
- Provee encapsulamiento en dos niveles: tipos de datos y objetos.

Arquitectura CORBA: IDL

```
// Ejemplo de especificación de IDL: mybank.idl
Module BANK {
      interface BankAccount {
          enum account_kind {checking,saving}; //types
          exception account_not_available {string reason}; //exceptions
          exception incorrect_pin{};
          readonly attribute float balance; //atributes
          attribute account_kind  what_kind_of_account;
          //operations
          void access (in string account, in string pin)
              raises (account_not_available, incorrect_pin);
          void deposit (in float f, out float new_balance)
              raises (account_not_available);
          void withdraw (in float f, out float new_balance)
              raises (account_not_available);
      }
}
```

Arquitectura CORBA: IDL

// Ejemplo de especificación de IDL: forma.idl y ListaForma.idl

```
struct Rectangulo {
    long ancho;
    long alto;
    long x;
    long y;
};

struct ObjetoGrafico{
    string tipo;
    Rectangle enmarcado;
   boolean estaRelleno;
};
```

```
interface Forma {
    long dameVersion();
    ObjetoGrafico DameTodoEstado();
};

Typedef sequence <Forma, 100> Todo
Interface ListaForma {
    exception ExceptionLlena{};
    Forma nuevaForma(in ObjetoGrafico g)
    raises (ExceptionLlena)
    Todo TodasFormas();
    long dameVersion();
};
```

Lista de Ref. a objetos CORBA

Arquitectura CORBA: IDL

- Módulos: la construcción módulo permite agrupar en unidades lógicas las interfaces y otras definiciones del tipo IDL. Define un alcance léxico.
- Los parámetros se etiquetan como *in*, *out*, *inout*.
  - *in*: se pasa del cliente al objeto CORBA invocado.
  - *out*: lo devuelve el objeto CORBA
  - *inout*: el valor de este parámetro puede pasarse en ambas direcciones.

Arquitectura CORBA: IDL

- Si no se devuelve ningún valor el tipo retornado se coloca como *void*.

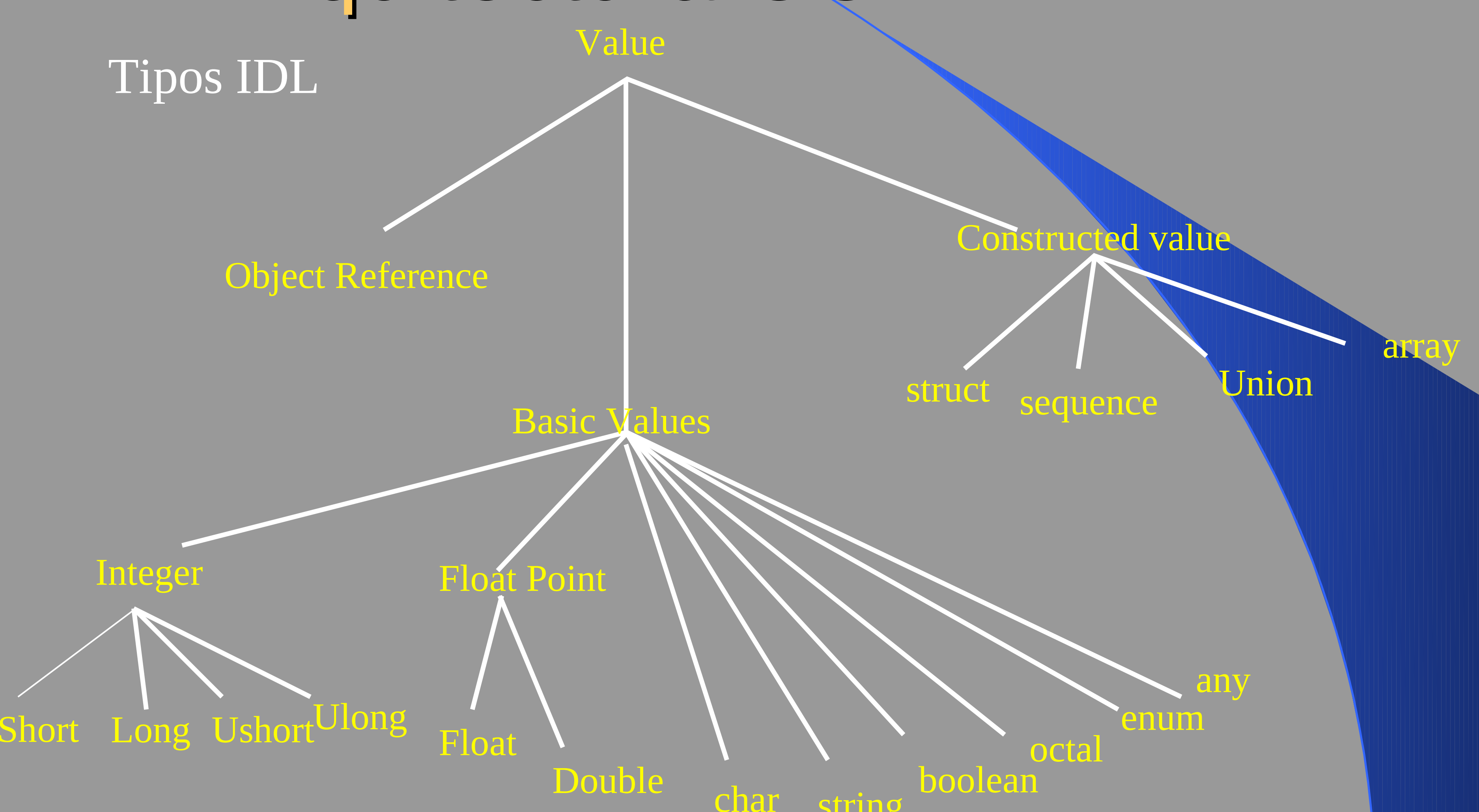
```
oneway void retrollamada (in int version)
```

- *Oneway* indica que el cliente que invoca el método no se bloqueará mientras el destino lleva a cabo el método.
- Cuando se lanza una excepción que contiene variables, el servidor puede utilizar las variables para devolver información al cliente sobre la excepción.

```
exception ExceptionLlena{};
exception ExceptionLlena{ObjetoGrafico g};
```

Arquitectura CORBA: IDL

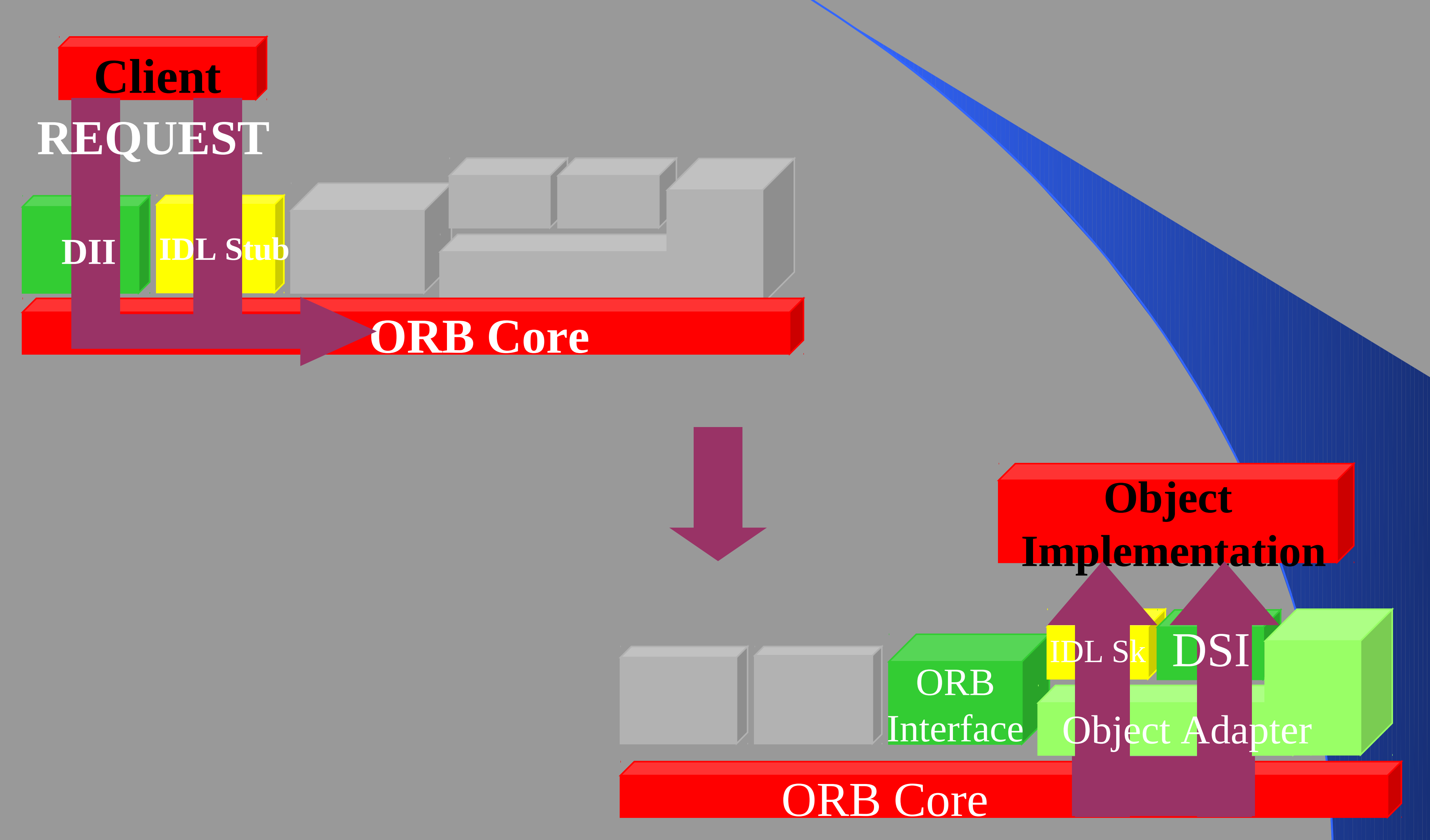
Tipos IDL



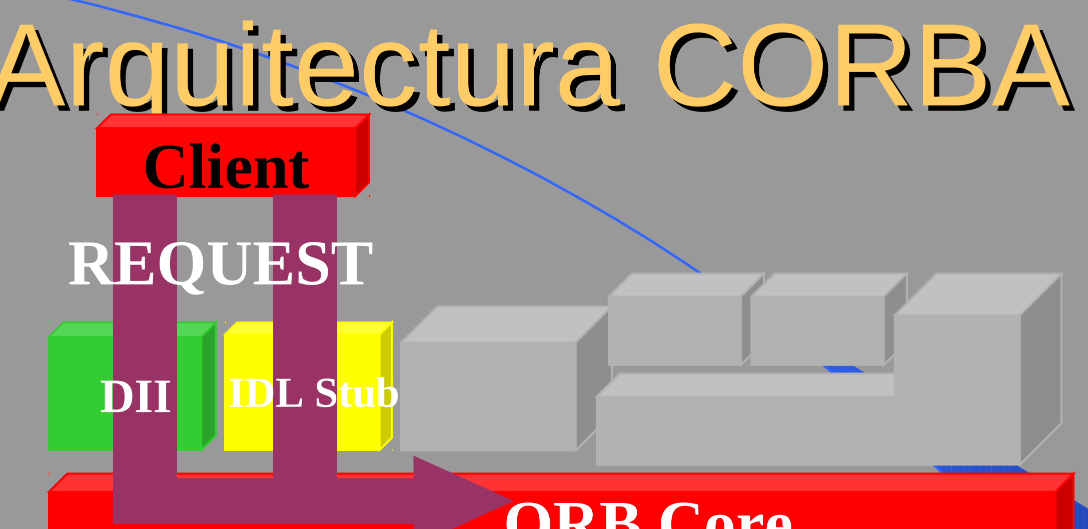
Arquitectura CORBA: IDL

- Herencia: las interfaces IDL se pueden extender.
- Una interfaz IDL podrá extender de más de una interfaz.
- Todas las interfaces IDL heredan del tipo Object. Esto hace posible el definir operaciones IDL que puedan tomar como argumento o devolver como resultado una referencia a un objeto remoto de cualquier tipo.

Arquitectura CORBA: IDL



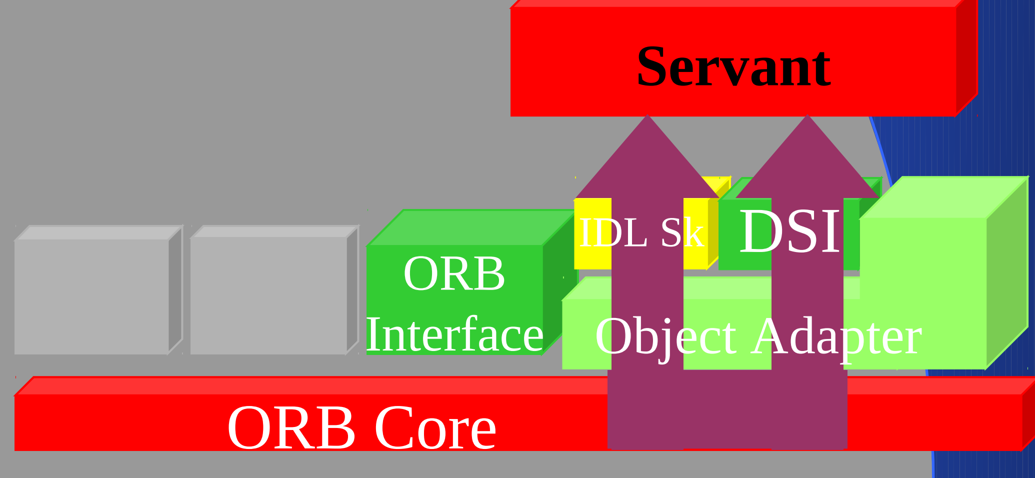
Arquitectura CORBA



- IDL Stubs (resguardos o proxies):
  - Funciones generadas desde la interfaz IDL para "enlazarlas" a los clientes
  - Provee una interfaz de invocación estática
- Dynamic Invocation Interface (DII)
  - Permite especificar y construir requerimientos a tiempo de ejecución
  - Operaciones: *create_request*, *invoke*, *send*, *get_response*
  - El cliente especifica el objeto, la operación y los parámetros. Se obtienen a través del repositorio de interfaz.

Arquitectura CORBA

- IDL Skeleton (Esqueletos):
  - Funciones generadas desde la interfaz IDL para “enlazarlas” a las implementaciones de objetos
- Dynamic Skeleton Interface (DSI)
  - Análogo al DII del lado de la implementación de objetos
  - Puede recibir invocaciones estáticas o dinámicas desde los clientes



Arquitectura CORBA

- ORB Interface:
  - Provee funciones para acceder directamente al ORB core desde los clientes y desde las implementaciones de objetos
  - Su interfaz no depende de la interfaz de los clientes ni de las interfaces de las implementaciones de objetos
  - Operaciones que permiten su arranque y parada
  - Operaciones para la conversión entre referencias a objetos remotos y cadenas de texto.
  - Operaciones para obtener listas de argumentos para llamadas con invocación remota

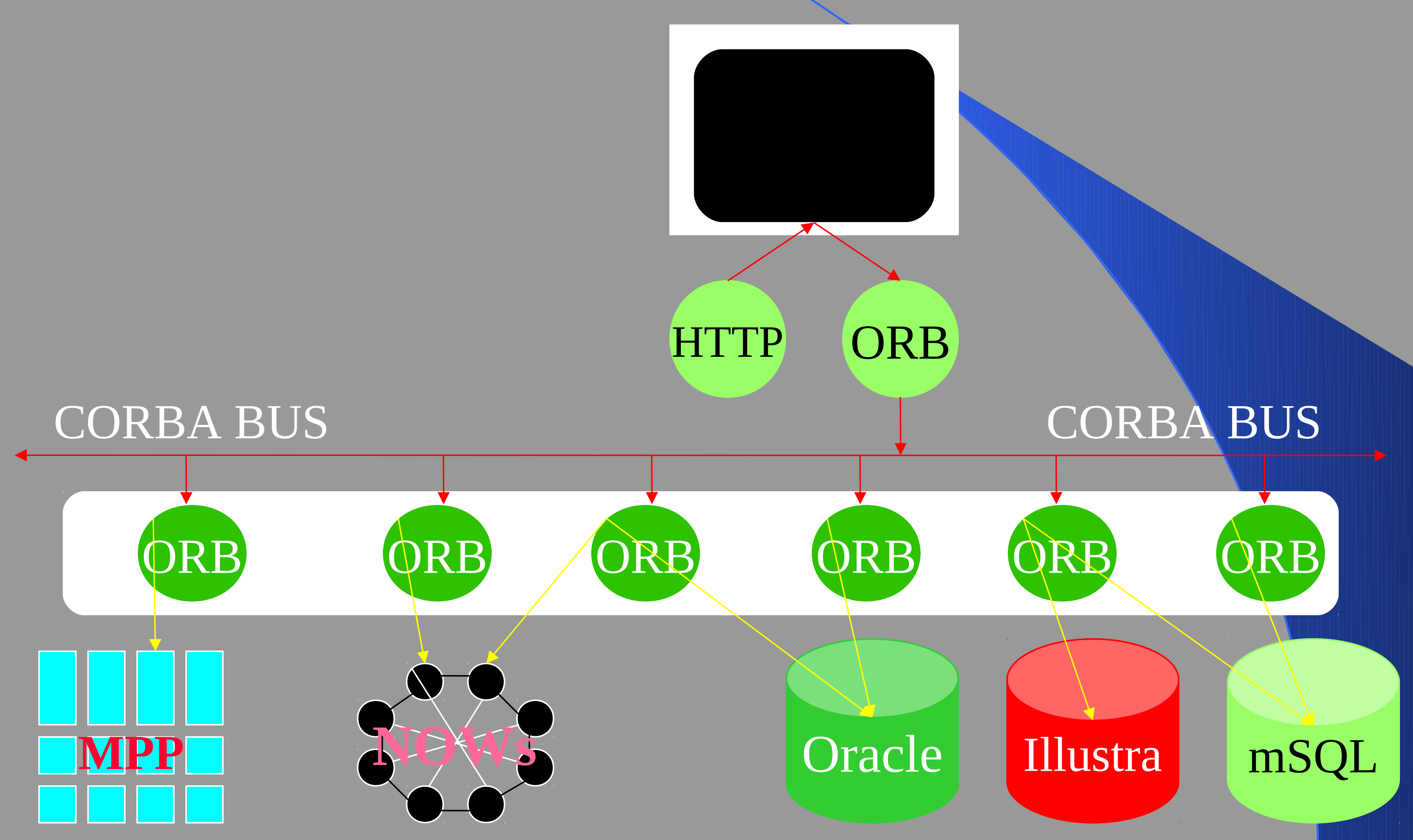
Arquitectura CORBA

- Object Adapter:
  - Provee funciones para instanciar objetos, pasar requerimientos y manipular referencias de objetos
  - Provee inter-operabilidad
  - Permite crear referencias remotas a los objetos CORBA
  - Despacha las llamadas al sirviente apropiado

Arquitectura CORBA

- Repositorio de Interfaces (*Interface Repository*):
  - Su información permite que un programa encuentre un objeto cuya interfaz no conoce en tiempo de compilación
- Repositorio de Implementaciones (*Implementation Repository*):
  - Contiene información que permite al ORB core localizar y activar implementaciones de objetos

Arquitectura CORBA



Estructura de un Adaptador de Objetos

Object Implementation

Interface A methods

Interface B methods

Dynamic Skeleton

Interface A Skeleton

Interface B Skeleton

Object Adapter Interface

ORB core

Estructura de un Adaptador de Objetos

- Tiene tres interfaces diferentes:
  - Una interfaz privada para el *esqueleto*
  - Una interfaz privada para el núcleo
  - Una interfaz pública para las implementaciones de objetos

Funciones de un Adaptador de Objetos

- Funciones:
  - Genera referencias a objetos que se registran en CORBA. IOR (Interoperate object reference)
  - Da a cada objeto CORBA un único nombre que forma parte de su referencia a objeto remoto.
  - Medio de comunicación entre implementaciones de objetos y el ORB core.
  - Despacha cada RMI vía un esqueleto hacia el sirviente apropiado.
  - Registro, Activación/Desactivación de *sirvientes*
  - Persistencia y administración de objetos compartidos

Funciones de un Adaptador de Objetos

Que pasaría si no existiera el adaptador de objetos?

- El core tendría que soportar múltiples tipos o estilos de sirvientes lo cual lo haría más grande, complejo y lento.
- Falta de flexibilidad

Con el OA se pueden soportar diferentes estilos de programación de los sirvientes. De esta forma el código del ORB es más pequeño y simple.

Estructura de un Adaptador de Objetos (BOA)

- Modelos de Activación
  - *Unshared Server*: servidor que soporta un único objeto CORBA. Si se crea otro objeto (del mismo tipo) se tiene que crear un nuevo servidor.
  - *Shared Server*: soporta múltiples objetos CORBA, posiblemente de diferentes tipos.
  - *Persistent Server*: el servidor se inicia cuando se inicia el sistema (boot time).
  - *Server-per-Method*: es una colección de procesos c/uno implementando una operación de un objeto CORBA.

Estructura de un Adaptador de Objetos

POA: facilita la portabilidad de los servidores CORBA.

- Identificador de objetos
- Objetos persistentes y transitorios
- Activación asociada al objeto, no al servidor: explícita, por demanda, implícita, por defecto
- Explicitamente se soportan servidores multithreaded.
- Existen otras diferencias (Schmidt and Vinosky)

Estructura de un Adaptador de Objetos (POA)

- Modelos de Activación
  - *Activación explícita*: el programador del servicio registra los sirvientes de objetos CORBA con llamadas explícitas sobre POA
  - *Activación por demanda*: el programador del servicio registra un administrador de sirvientes y POA upcalls cuando recibe un requerimietno: encarnación de sirvientes, forwardrequest a otro sirviente, etc.)
  - *Activación implícita*: se activa sin llamada explícita sobre POA.
  - *Sirviente por defecto*: la aplicación registra un sirviente por defecto que es usado ante un request cuyo objeto CORBA no está activado, no hay administrador de sirvientes (encarnación de todos los objetos CORBA).

Inter-Operabilidad

- Es necesario que inter-operen los ORB´s de diversos fabricantes: GIOP y la referencia a objetos uniforme
- El protocolo General Inter-ORB (GIOP) satisface las necesidades de comunicación entre ORB´s y trabaja sobre cualquier protocolo de transporte.
- La implementación del GIOP que funciona sobre TCP/IP se denomina *Internet Inter-ORB Protocol* (IIOP)

Referencia a Objetos

- IOP: Interoperable Object Reference

Un objeto CORBA puede identificarse, localizarse y direccionarse por su referencia a objeto.

Nombre de tipo de Interfaz IDL	Protocolo y dirección detallada			Clave del Objeto	
Identificador de repositorio de interfaz	IIOP	Nombre de Dominio del host	Número de puerto	Nombre del adaptador	Nombre o ID del objeto

Modelos de Interacción de CORBA

- La invocación remota en CORBA define, por defecto, la semántica *como máximo una vez*. Si una operación retorna exitosamente, se garantiza que se ejecutó exactamente 1 vez. Si retorna una excepción fue ejecutada a lo sumo 1 vez.

Modelos de Interacción de CORBA

- Síncronas: RPC, protocolo *at-most-once*, soporta invocaciones estáticas y dinámicas
- Asíncronas: semánticas de operaciones *one-way y best-effort*, soporta invocaciones estáticas y dinámicas
- Síncrona diferida: protocolo *at-most-once*, *sólo* soporta invocaciones dinámicas

Implementaciones de ORBs

- Residente en el cliente y en la implementación de objeto
- Basado en un Servidor
- Basado en el Sistema
- Basado en Librerías

Servicios y Facilidades

- Objetos de aplicaciones: resuelven problemas particulares de usuarios particulares
- CORBA Facilities: Servicios de nivel intermedio. Verticales y Horizontales
- CORBA Services: Servicios básicos para aplicaiones
- CORBA core e interoperabilidad

Servicios y Facilidades

- CORBA Services:
  - Ciclo de vida de los objetos
  - Manejo de nombres
  - Persistencia
  - Notificación de eventos
  - Concurrencia y transacciones
  - Seguridad
  - Servicio de tiempo

Servicios y Facilidades

- CORBA Facilities:
  - Horizontales: facilidades comunes
    - Interfaz con el usuario
    - Administración de la información
    - Administración del sistema
    - Administración de tareas

Servicios y Facilidades

- CORBA Facilities:
  - Verticales: facilidades particulares
    - Financieras
    - De telecomunicaciones
    - Audio y video
    - Petroquímica

Paradigmas de Computación Distribuida

- Objetos distribuidos (Distributed Object Computing). Tiene todas las ventajas de la programación orientada por objetos.
- Desarrollo basado en componentes (Component-based development): Industrialización del desarrollo de software

Software orientado a componentes

- Representa la industrialización del desarrollo de software.
- En la industria electrónica como en otras industrias se usan componentes para construir placas, tarjetas, etc.
- En el campo del software la idea es la misma. Se puede crear una interfaz de usuario en un programa en base a componentes: páneles, botones, menúes, etc.

Software orientado a componentes

- Los componentes son elementos de software autocontenidos que pueden controlarse en forma dinámica y ensamblarse para construir aplicaciones
- Funcionan de acuerdo a un conjunto de reglas y especificaciones.
- Proveen cierta funcionalidad que puede ser reutilizada en diferentes lugares

Software orientado a componentes

- Con la utilización de componentes se gana en calidad y rapidez de desarrollo.
- Los Java beans (oct. 1996) constituyen la arquitectura de componentes de Java (independientes de la platafotma) y su uso ha probado ser de mucho valor en el desarrollo de aplicaciones *network-aware*
- Otros componentes: VBX (Visual Basic Extension), ActiveX (Microsoft), OpenDoc (Apple-IBM) y otros (No son portables)

Orientación por objetos vs. componentes

Proponents of object-oriented programming (OOP) maintain that software should be written according to a mental model of the actual or imagined objects it represents. OOP and the related disciplines of object-oriented analysis and object-oriented design focus on modeling real-world[citation needed] interactions and attempting to create "nouns" and "verbs" that can be used in more human-readable ways, ideally by end users as well as by programmers coding for those end users.

Component-based software engineering, by contrast, makes no such assumptions, and instead states that developers should construct software by gluing together prefabricated components - much like in the fields of electronics or mechanics. Some peers will even talk of modularizing systems as software components as a new programming paradigm.

JavaBeans: Un caso de estudio

¿Qué es JavaBeans?

- Es una capacidad que se inicia con core JDK 1.1 (96-97). Actualmente es la arquitectura de componentes de Java 2 Platform, Standard Edition (J2SE)
- Es un ejemplo de:
  - Arquitectura basada en componentes portables y reusables
  - Comunicación a través de eventos
  - Hace posible escribir componentes WORA *(write once run anywhere)* en Java

¿Qué es JavaBeans?

- Java provee la reusabilidad por ser orientado por objetos y Javabeans extiende este potencial con especificaciones que permiten unir componentes para armar aplicaciones
- Dirigido a personas que no son programadores, posiblemente expertas en el negocio.

¿Qué es JavaBeans?

- Arquitectura de plataforma neutral para ambientes de aplicaciones Java
- Ideal para desarrollar soluciones para ambientes con Sistemas de Operación y hardware heterogéneos

¿Qué es JavaBeans?

- Componentes de software reutilizables que se puedan manipular visualmente en una herramienta de programación visual o a través de un programa.
- Para ello se define una interfaz que permite a la herramienta de programacion o IDE (*Integrated Development Environment*) interrogar al bean para conocer las propiedades que define y los tipos de eventos que puede generar en respuesta a diversas acciones.

¿Qué es JavaBeans?

- Pueden ser usados como parte de una estrategia de computación distribuida. Los beans se pueden distribuir en diversas máquinas y servidores
- Son componentes autocontenidos, unidades de software reusables que pueden ser visualmente compuestas en *applets* o aplicaciones

¿Qué es JavaBeans?

- Los beans deben residir dentro de un *Container*
- *El container* provee el ambiente necesario para diseñar y crear aplicaciones basadas en Beans. Ofrece facilidades de comunicación entre beans
- Los beans pueden ser *Containers* de otros beans
- Dentro del *Container* se puede seleccionar un Bean, modificar su apariencia y comportamiento, definir interacción con otros Beans y ensamblarlos en una aplicación, un *applet* o un nuevo Bean.

¿Qué es JavaBeans?

- A pesar de haber muchas semejanzas, los JavaBeans no deben confundirse con los Enterprise JavaBeans (EJB), una tecnología de componentes del lado servidor que es parte de Java EE.

- Los Java Beans son una tecnología dirigida a la construcción de clientes.

Convenciones de Beans

- Las convenciones requeridas son:
  - La clase debe ser serializable (capaz de salvar persistentemente y de restablecer su estado)
  - Debe tener un constructor sin argumentos
  - Sus propiedades deben ser accesibles mediante métodos get y set que siguen una convención de nomenclatura estándar
  - Debe contener determinados métodos de manejo de eventos

Ejemplo

```
public class PersonBean implements
java.io.Serializable {
  private String name;
  private int age;
  public PersonBean() {
     // Constructor sin argumentos
  }
  public void setName(String n) {
    this.name = n;
  }
  public void setAge(int a) {
    this.age = a;
  }
  public String getName() { return
(this.name); }
  public int getAge() { return
(this.age); }
}
```

Ejemplo

```
PersonBean person = new PersonBean();
person.setName("Roberto");
System.out.println(person.getName());


}
```

Características Básicas

- Propiedades: atributos que afectan su apariencia o conducta
- Eventos: mecanismo usado por los componentes para enviar notificaciones a otros componentes
- Personalización: posibilidad de alterar la apariencia y conducta del bean a tiempo de diseño y de ejecución. Esto es necesario si se quiere que el bean sea re-utilizado.

Características Básicas

- Persistencia: posibilidad de guardar y recuperar el estado de los beans, a través de la serialización. El programador decide cuáles propiedades son persistentes y cuáles transitorias.

- Introspección: Permite a la herramienta de programación o IDE, analizar el comportamiento del bean (exposición de propiedades y métodos)

Propiedades de los Beans

- Son los atributos de apariencia y comportamiento que pueden cambiar en el momento del diseño. Por ejemplo, un botón puede tener las siguientes propiedades: tamaño, posición, título, color de fondo, color de texto, si está o no habilitado, etc.

- Un IDE sabe cómo analizar un bean y conocer sus propiedades.

Propiedades de los Beans

- El IDE crea una representación visual para cada uno de los tipos de propiedades, denominada editor de propiedades, para que el programador pueda modificarlas fácilmente en el momento del diseño.

- Los beans deben seguir un conjunto de convenciones de nombre para que el IDE pueda inferir qué métodos corresponden a qué propiedades.

Propiedades de los Beans

- Cuando se selecciona un bean en el panel, aparece una hoja de propiedades (lista de las propiedades del bean), con los editores asociados a cada una de las propiedades.
- El IDE llama a los métodos que empiezan por **get**, para mostrar en los editores los valores de las propiedades.
- Si el programador cambia el valor de una propiedad se llama al correspondiente método **set**, para actualizar el valor de dicha propiedad (esto puede o no afectar al aspecto visual del bean en el momento del diseño)

Propiedades de los Beans

- Propiedades públicas: accesibles por otros beans y pueden tener asociado métodos acesores y métodos mutadores
- Propiedades privadas: son internas y escondidas

Propiedades de los Beans

- Propiedades simples: representan un único valor:

```
//nombre del atributo que se usa para
//guardar el valor de la propiedad
private String nombre;

//métodos set y get de la propiedad
public void setNombre(String nuevoNombre) {
     nombre=nuevoNombre;
}
public void getNombre() {
     return nombre;
}
```

Propiedades de los Beans

- Propiedades simples: propiedad booleana

```
//miembro de la clase que se usa para guardar
// el valor de la propiedad
private boolean conectado=false;
//métodos set y get de la propiedad
// denominada Conectado
public void setConectado(boolean nuevoValor){
    conectado=nuevoValor;
}
public boolean isConectado(){
    return conectado;
}
```

Propiedades de los Beans

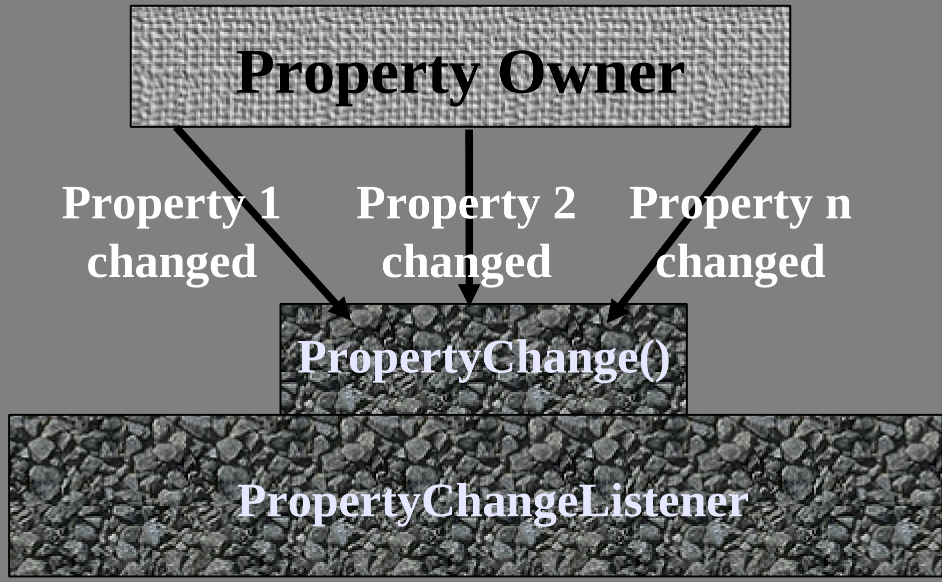
- Propiedades indexadas: tienen asociado más de un valor (arreglos, vectores)

```
private int [] numeros=[1,2,3,4];
//métodos get y set de
  la propiedad
public void setNumeros(int[]
nuevoValor); {
      numeros=nuevoValor;
}
public int [] getNumeros(){
      return numeros;
}
```

```
//metodos get y set para
  los elementos
public void setNumeros
(int ind, int nuevoValor) {
  numeros[ind]=nuevoValor;
}
public int getNumeros(int
ind) {
   return numeros[ind];
}
```

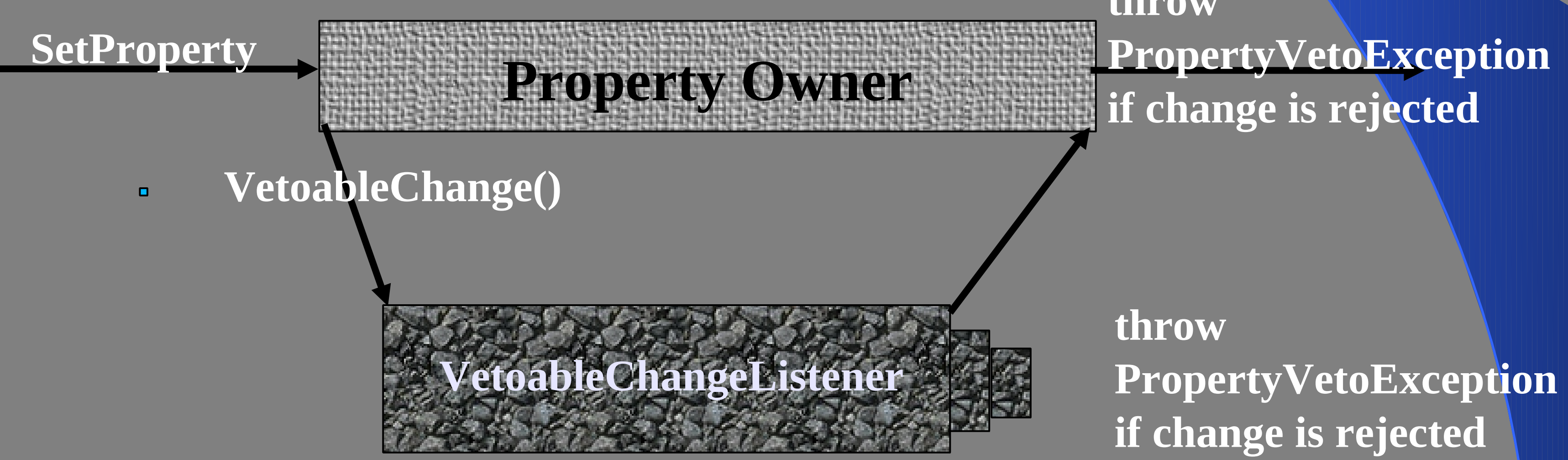
Propiedades de los Beans

- Propiedades ligadas (*bound*): asociadas a eventos. Cuando se modifica la propiedad se dispara un evento para notificar a otros beans de tal cambio



Propiedades de los Beans

- Propiedades restringidas (*constrained)*: similares a las propiedades *bound* salvo que el Bean notificado debe validar tal modificación y puede vetar el cambio.



Eventos

- Eventos: se usan para comunicarse con otros beans. Se requiere de objetos fuentes (*sources*) que disparan el evento y objetos escuchadores (*listeners*) que manejan el evento (modelo de delegación de eventos)
- Muchos objetos pueden estar esperando un determinado evento

Eventos

- El generador del evento envia la notificación de su ocurrencia mediante la invocación de un método en el *listener*. Se usan métodos estándares de Java.

- Cuando se llama al método se pasa un objeto como párametro que contiene información sobre el evento ocurrido

Personalizadores (Customizers)

- Grado de facilidad que el componente tiene de “acomodar” requerimientos de aplicaciones diferentes.
- Los Beans deben proveer mecanismos para personalizarlos, permitiendo alterar su apariencia y comportamiento
- Los Personalizadores permiten editar varias propiedades a la vez, junto con o en lugar de los Editores de Propiedades

Personalización: Editores de Propiedades

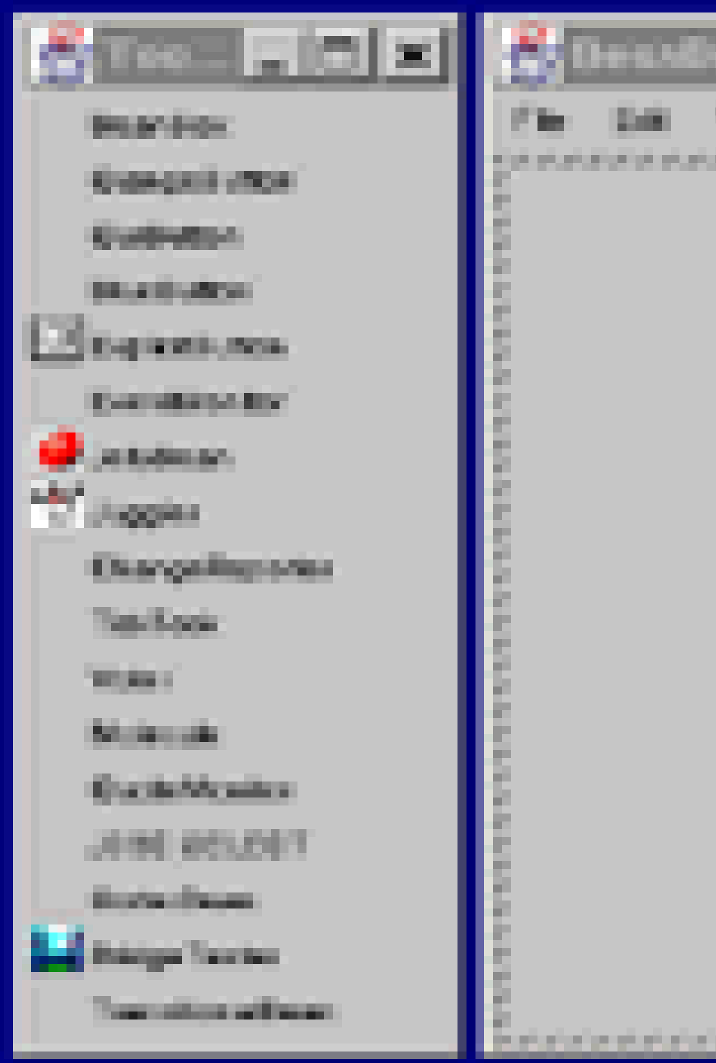
- En *ToolBox* existe una “hoja de propiedades” para cambiar los valores de las propiedades del Bean que se esté editando.
- Se requieren de Editores de Propiedades
- El *BeanBox* trae ciertos editores de propiedades básicos (color, strings y boleanos)
- Se pueden sustituir o agregar editores de propiedades con el *Property Editor Manager*

Personalización: Ventana de BeanBox

Tool Box

BeanBox

Property Sheet



Persistencia

- Capacidad de guardar el estado de un Bean junto con los valores personalizados (propiedades persistentes y transitorias)
- Es necesario que los beans implementen la interfaz `Serializable`.
- Las propiedades que no se requiera que sean serializables deben marcarse como `transcient`

Persistencia

- Ejemplo:

```
Class MyBeanClass implements Serializable {
    transient Thread t;
    // constructor sin argumentos
    public myBean {...}
}
```

- Un Bean debe tener por lo menos un constructor sin argumento.
- Una vez que se ha desarrollado un Bean se requiere empaquetarlo como un archivo `.jar` y colocarlo en el subdirectorio `jars` del directorio `beanbox`

Introspección

- Proceso por el cual una herramienta constructora u otro bean descubre las propiedades, métodos y eventos asociados a un Bean
- Se puede realizar de dos formas:
  - Buscando las clases y métodos que siguen ciertos patrones de nombres (usando reflexión de Java)
  - Realizando *queries* al BeanInfo de una clase

Introspección: Patrones de Diseño

- Patrones de nombres estándares para las propiedades y los eventos de un Bean
- Se usa con el mecanismo de reflexión de Java para descubrir las propiedades de los Beans

```
import java.lang.reflect.*;

    public class DumpMethods {
        public static void main(String args[]) {
            try {
                Class c = Class.forName(args[0]);
                Method m[]=c.getDeclareMethods();
                for (int i=0;i<m/length;i++)
                    System.out.println(m[i].toString());
            }
            catch (Throwable e) {
                System.err.println(e);
            }
        }
    }
```

Para una invocación de:

```
$> java DumpMethods java.util.Stack
```

La salida es:

```
public java.lang.Object.java.util.Stack.push(java.lang.Object)
public synchronized java.lang.Object.java.util.Stack.pop()
public synchronized java.lang.Object.java.util.Stack.peek()
public boolean java.util.Stack.empty()
public synchronized int java.util.Stack.search(java.lang.Object)
```

Introspección: Patrones de Diseño

- Los métodos asesores/mutadores deben comenzar con la palabra “get”/”set” en minúsculas seguido del nombre de la propiedad con la primera letra en mayúscula.

```
public X  getPropertyName()
public void setPropertyName(X x)
```

Corresponden a los métodos asesor y mutador de la propiedad de tipo X

- `public X getPropertyName()`

Sin el correspondiente método “set”, indica una propiedad *readonly*

Introspección: Patrones de Diseño

- `public bool isPropertyName()`
  `public void setPropertyName (boolean b)`

- Corresponden a los métodos asesor y mutador de una propiedad booleana.

- En general los atributos *strings* o números sólo usan métodos `set/get`

Introspección: Patrones de Diseño

- ```
  Public X] getPropertyName()
  public void setPropertyName (X[] x)
  public X getPropertyName(int i)
  public void setPropertyName (int i, X x)
  ```

  Corresponden a los métodos asesores y mutadores de una propiedad indexada.
  ```

Introspección: Patrones de Diseño

- Propiedades “Bound”: avisan a los *listeners* interesados que su valor ha cambiado. Se deben implementar dos mecanismos:
  - Si el valor de la propiedad cambia se genera el evento `PropertyChangeEvent` a todos los *listeners* registrados. El evento puede ocurrir cuando el método `set` es invocado.
  - Para permitir que los *listeners* se registren, el Bean tiene que implementar los siguientes métodos:

```
public void addPropertyChangeListener
  (PropertyChangeListener l)
public void removeChangeListener
     (PropertyChangeListener l)
```

Introspección: Patrones de Diseño

- Propiedades "Constrained": La idea es quen éstas sean propiedades de un Bean tal que ciertos cambios de estado puedan ser vetados por diferentes usuarios del Bean. Para construir propiedades vetables, un Bean debe definir los siguientes métodos:
  - `public void addVetableChangeListener (VetableChangeListener l)`
  - `public void removeVetableChangeListener (VetableChangeListener l)`
  - `private void VetableChangeSupport()`

Si el cambio es vetado el *listener* indica su desaprobación con `PropertyVetoException`

Introspección: Patrones de Diseño

- Manejo de eventos
  - Los nombres de los eventos deben seguir el formato `EventNameEvent` (ejemplo `TimerEvent`)
  - La interfaz del *listener* debe seguir el formato `EventNameListener` (ejemplo `TimerListener`)

1. El Bean debe implementar los métodos:

```
public void addEventNameListener
   (EventNameListener e)
public void removeEventNameListener
   (EventNameListener e)
```

Introspección: Patrones de Diseño

- Manejo de eventos (continuación)
  - Un Bean puede responder a otros eventos que no implican la modificación del valor de un atributo (*custom events)*
  - Definir el evento: `Class CustomEvent extends EventObject`
  - Escribir una interfaz `CustomListener` con un método de notificación

```
public void addCustomListener
   (CustomListener e)
public void removeCustomListener
   (CustomListener e)
```

Introspección: Clase BeanInfo

- Mecanismo flexible y poderoso para almacenar información sobre los Beans. Esta información es usada por los constructores de Beans para Instropección.
- Es un mecanismo alterno a la Reflexión. Se usa cuando los patrones de diseño no abarcan características más complejas de los Beans
- La idea es implementar una interfaz llamada `BeanInfo` que va a ser interrogada por el constructor de Beans

Introspección: Clase BeanInfo

- El nombre de la clase BeanInfo asociada a un Bean debe seguir el modelo:

```
Class NameBeanBeanInfo implements
            SimpleBeanInfo
```

- Es necesario crear descriptores de características (eventos, propiedades y métodos)

Introspección: Clase BeanInfo

- Es necesario crear métodos que retornen los descriptores de las características:

```
EventSetDescriptor[] getEventSetDescriptors()
MethodDescriptor[] getMethodDescriptors()
PropertyDescriptor[] getPropertyDescriptors()
```

Se usa `instanceof` para chequear si una `PropertyDescriptor` específica es una `IndexedPropertyDescriptor`

¿Cómo hacer los Beans reusables?

- Los Beans están dirigidos a personas no programadoras, por lo tanto deben ser manipilables a través de herramientas de diseño visual sin usar lenguajes de programación
- Los beans pueden ser usados por expertos programadores, por lo tanto hay que proveer diferentes comportamientos

¿Cómo hacer los Beans reusables?

- ¿Qué clases de Java pueden ser Beans?
- ¿Puede ser usada por más de un área?¿Otros se benefician al usarla?
- ¿Existen diferentes formas de personalizarla?
- ¿Es fácil explicar su propósito?
- ¿Contiene toda la información necesaria para manejarla?¿Tiene buen encapsulamiento?

¿Cómo usar y probar Beans?

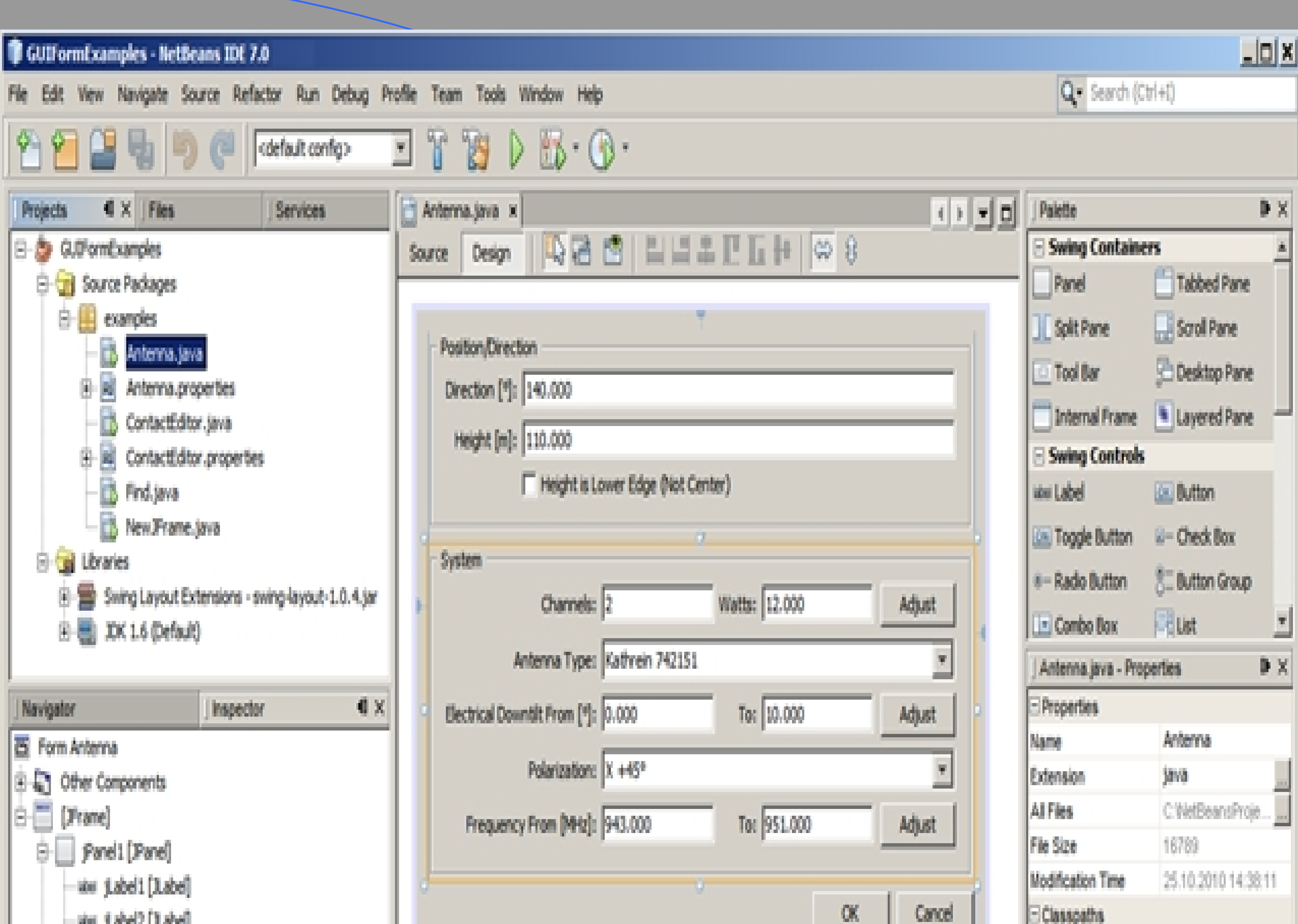
- El **BeanBox** es parte del Bean Developer Kit de JavaSoft. Es una herramienta constructora de aplicaciones
- Editores de propiedades: para cambiar los valores por defecto de las propiedades de los beans
- Editores de propiedades especializados: provistos por los desarrolladores de los Beans

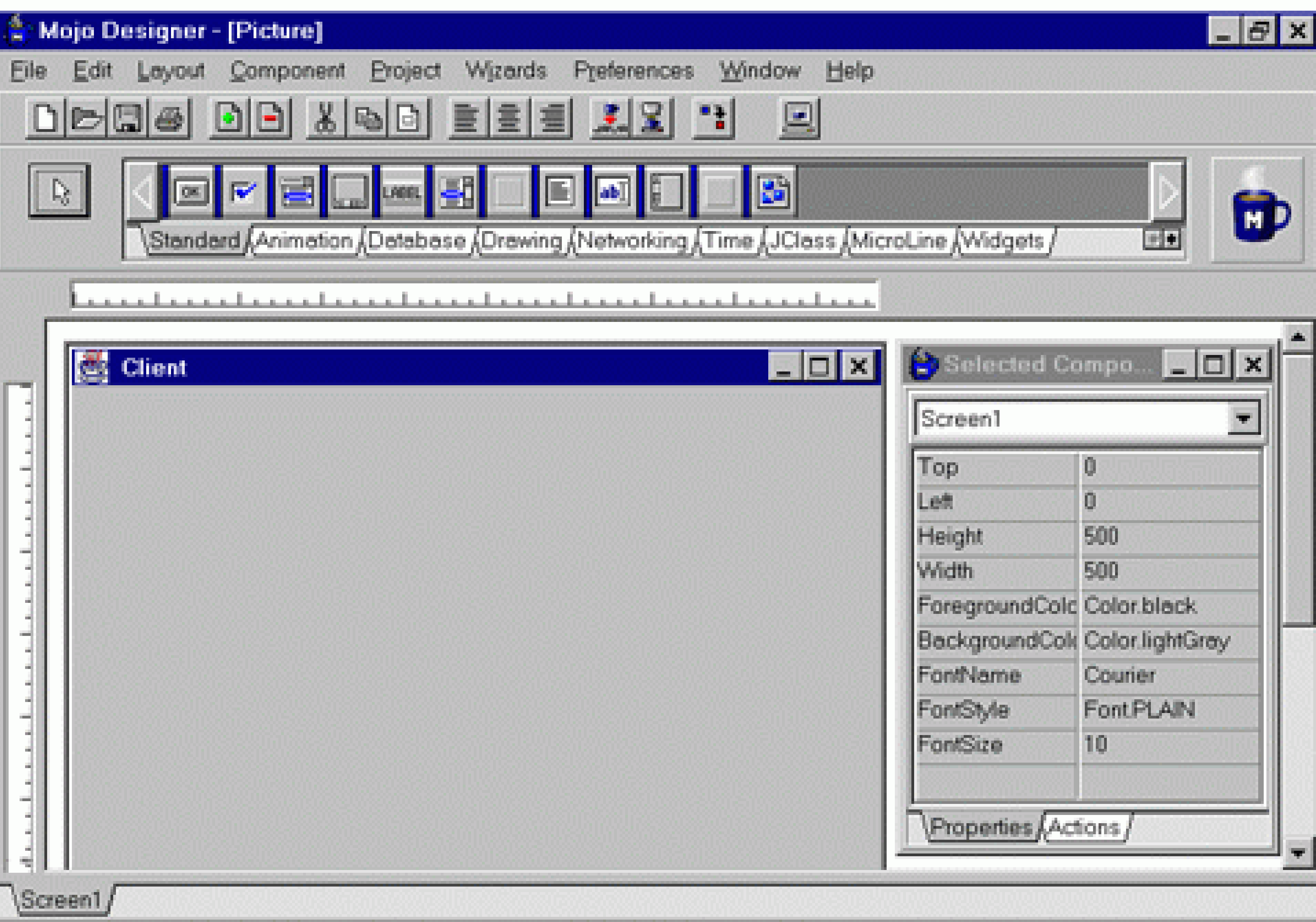
¿Cómo usar y probar Beans?

- Beans visuales extienden de *Component*
- Beans no visuales extienden de *Object*
- Para agregar un Bean al *ToolBox* del BeanBox es necesario crear un archivo `.jar` y colocarlo en el subdirectorio `jars` del directorio `beanbox`

¿Cómo usar y probar Beans?

- Una aplicación se crea en un IDE seleccionando los componentes visibles e invisibles en una paleta de herramientas y situarlos sobre un panel o una ventana.
- Con el ratón se unen los sucesos (events) que genera un objeto (fuente), con los objetos (listeners) interesados en responder a las acciones sobre dicho objeto.


GUIFormExamples - NetBeans IDE 7.0
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
Projects
Files
Services
GUIFormExamples
Source Packages
examples
Antenna.java
Antenna.properties
ContactEditor.java
ContactEditor.properties
Find.java
NewJFrame.java
Libraries
Swing Layout Extensions - swing-layout-1.0.4.jar
JDK 1.6 (Default)
Navigator
Inspector
Form Antenna
Other Components
[JFrame]
jPanel1 [JPanel]
jLabel1 [JLabel]
jLabel2 [JLabel]
Antenna.java
Source
Design
Position/Direction
Direction [°]: 140.000
Height [m]: 110.000
Height is Lower Edge (Not Center)
System
Channels: 2
Watts: 12.000
Adjust
Antenna Type: Kathrein 742151
Electrical Downtilt From [°]: 0.000
To: 10.000
Adjust
Polarization: X +45°
Frequency From [MHz]: 943.000
To: 951.000
Adjust
OK
Cancel
Palette
Swing Containers
Panel
Tabbed Pane
Split Pane
Scroll Pane
Tool Bar
Desktop Pane
Internal Frame
Layered Pane
Swing Controls
Label
Button
Toggle Button
Check Box
Radio Button
Button Group
Combo Box
List
Antenna.java - Properties
Properties
Name
Antenna
Extension
java
All Files
File Size
Modification Time
Classpaths



Mojo Designer - [Picture]
File Edit Layout Component Project Wizards Preferences Window Help
Standard Animation Database Drawing Networking Time JClass MicroLine Widgets
Client
Selected Compo...
Screen1
Top 0
Left 0
Height 500
Width 500
ForegroundColc Color.black
BackgroundColc Color.lightGrey
FontName Courier
FontStyle Font.PLAIN
FontSize 10
Properties Actions
Screen1
Picture Component

Selected Compo...

Screen1

Top	0
Left	0
Height	500
Width	500
ForegroundColo	Color.black
BackgroundColo	Color.lightGray
FontName	Courier
FontStyle	Font.PLAIN
FontSize	10

Properties Actions

Object In...

Form1: TForm1

ActiveControl	
AutoScroll	True
+BorderIcons	[biSystemMenu,I
BorderStyle	bsSizeable
Caption	Form1
ClientHeight	269
ClientWidth	427
Color	clBtnFace
Ctl3D	True
Cursor	crDefault
Enabled	True
+Font	(TFont)
FormStyle	fsNormal
Height	300
HelpContext	0

Properties Events